Phụ lục 5

TRƯỜNG ĐẠI HỌC TRÀ VINH
KHOA KỸ THUẬT VÀ CÔNG NGHỆ
BỘ MÔN CÔNG NGHỆ THÔNG TIN

TÀI LIỆU GIẢNG DẠY

MÔN CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT 2

GV biên soạn: ***Phan Quốc Nghĩa***
Lê Minh Tự

Trà Vinh, 2013

Lưu hành nội bộ

MỤC LỤC

Nội dung **Trang**

CHƯƠNG 1
CẤU TRÚC DỮ LIỆU TẬP TIN

❖ **Mục tiêu học tập:** Sau khi học xong chương này, người học có thể nắm vững và vận dụng:
- Một số khái niệm về tập tin.
- Các bước thao tác với tập tin.
- Một số hàm truy xuất tập tin văn bản.
- Một số hàm truy xuất tập tin nhị phân.

.

1.1. MỘT SỐ KHÁI NIỆM VỀ TẬP TIN

Đối với các kiểu dữ liệu ta đã biết như kiểu số, kiểu mảng, kiểu cấu trúc thì dữ liệu được tổ chức trong bộ nhớ trong (RAM) của máy tính nên khi kết thúc việc thực hiện chương trình thì dữ liệu cũng bị mất; khi cần chúng ta bắt buộc phải nhập lại từ bàn phím. Điều đó vừa mất thời gian vừa không giải quyết được các bài toán với số liệu lớn. Để giải quyết vấn đề, người ta đưa ra kiểu tập tin (file) cho phép lưu trữ dữ liệu ở bộ nhớ ngoài (đĩa). Khi kết thúc chương trình thì dữ liệu vẫn còn do đó chúng ta có thể sử dụng nhiều lần. Một đặc điểm khác của kiểu tập tin là kích thước lớn với số lượng các phần tử không hạn chế (chỉ bị hạn chế bởi dung lượng của bộ nhớ ngoài).

Có 3 loại dữ liệu kiểu tập tin:

Tập tin văn bản (Text File): là loại tập tin dùng để ghi các ký tự lên đĩa, các ký tự này được lưu trữ dưới dạng mã Asc2. Điểm đặc biệt là dữ liệu của tập tin được lưu trữ thành các dòng, mỗi dòng được kết thúc bằng ký tự xuống dòng (new line), ký hiệu '\n'; ký tự này là sự kết hợp của 2 ký tự CR (Carriage Return - Về đầu dòng, mã Ascii là 13) và LF (Line Feed - Xuống dòng, mã Ascii là 10). Mỗi tập tin được kết thúc bởi ký tự EOF (End Of File) có mã ASCII là 26 (xác định bởi tổ hợp phím Ctrl + Z). Tập tin văn bản chỉ có thể truy xuất theo kiểu tuần tự.

Tập tin định kiểu (Typed File): là loại tập tin bao gồm nhiều phần tử có cùng kiểu: char, int, long, cấu trúc… và được lưu trữ trên đĩa dưới dạng một chuỗi các byte liên tục.

Tập tin không định kiểu (Untyped File): là loại tập tin mà dữ liệu của chúng gồm các cấu trúc dữ liệu mà người ta không quan tâm đến nội dung hoặc kiểu của nó, chỉ lưu ý đến các yếu tố vật lý của tập tin như độ lớn và các yếu tố tác động lên tập tin mà thôi.

Biến tập tin: là một biến thuộc kiểu dữ liệu tập tin dùng để đại diện cho một tập tin. Dữ liệu chứa trong một tập tin được truy xuất qua các thao tác với thông số là biến tập tin đại diện cho tập tin đó.

Con trỏ tập tin: Khi một tập tin được mở ra để làm việc, tại mỗi thời điểm, sẽ có một vị trí của tập tin mà tại đó việc đọc/ghi thông tin sẽ xảy ra. Người ta hình dung có một con trỏ đang chỉ đến vị trí đó và đặt tên nó là con trỏ tập tin. Sau khi đọc/ghi xong dữ liệu, con trỏ sẽ chuyển dịch thêm một phần tử về phíacuối tập tin. Sau phần tử dữ liệu cuối cùng của tập tin là dấu kết thúc tập tin EOF (End Of File).

1.2. CÁC THAO TÁC TRÊN TẬP TIN

Muốn thao tác trên tập tin, ta phải lần lượt làm theo các bước:
- Khai báo biến tập tin.

- Mở tập tin bằng hàm `fopen()`.
- Thực hiện các thao tác xử lý dữ liệu của tập tin bằng các hàm đọc/ghi dữ liệu.
- Đóng tập tin bằng hàm `fclose()`.

Ở đây, ta thao tác với tập tin nhờ các hàm được định nghĩa trong thư viện `stdio.h`.

1.2.1. Khai báo biến tập tin

Cú pháp:

`FILE` <Danh sách các biến con trỏ>

Các biến trong danh sách phải là các con trỏ và được phân cách bởi dấu phẩy(,).

Ví dụ: `FILE *f1,*f2;`

1.2.2. Mở tập tin

Cú pháp:

```
FILE *fopen(char *Path, const char *Mode)
```

Trong đó:

- Path: chuỗi chỉ đường dẫn đến tập tin trên đĩa.
- Mode: chuỗi xác định cách thức mà tập tin sẽ mở, các giá trị có thể của `Mode`:

Mode	Ý nghĩa
r	Mở tập tin văn bản để đọc
w	Tạo ra tập tin văn bản mới để ghi
a	Nối vào tập tin văn bản
rb	Mở tập tin nhị phân để đọc
wb	Tạo ra tập tin nhị phân để ghi
ab	Nối vào tập tin nhị phân
r+	Mở một tập tin văn bản để đọc/ghi
w+	Tạo ra tập tin văn bản để đọc ghi
a+	Nối vào hay tạo mới tập tin văn bản để đọc/ghi
r+b	Mở ra tập tin nhị phân để đọc/ghi
w+b	Tạo ra tập tin nhị phân để đọc/ghi
a+b	Nối vào hay tạo mới tập tin nhị phân

- Hàm fopen trả về một con trỏ tập tin. Chương trình của ta không thể thay đổi giá trị của con trỏ này. Nếu có một lỗi xuất hiện trong khi mở tập tin thì hàm này trả về con trỏ NULL.

Ví dụ: Mở một tập tin tên TEST.txt để ghi.

```
FILE *f;
f = fopen("TEST.txt", "w");
if (f!=NULL)
{
 /* Các câu lệnh để  thao tác với tập tin*/
 /* Đóng tập tin*/
}
```

Trong ví dụ trên, chúng ta có sử dụng câu lệnh kiểm tra điều kiện để xác định mở tập tin có thành công hay không?.

Nếu mở tập tin để ghi, nếu tập tin đã tồn tại thì tập tin sẽ bị xóa và một tập tin mới được tạo ra. Nếu chúng ta muốn ghi nối dữ liệu, chúng ta phải sử dụng chế độ "a". Khi mở tập tin để đọc, tập tin phải tồn tại nếu không một lỗi sẽ xuất hiện.

1.2.3. Đóng tập tin

Hàm `fclose()` được dùng để đóng tập tin được mở bởi hàm `fopen()`. Hàm này sẽ ghi dữ liệu còn lại trong vùng đệm vào tập tin và đóng lại tập tin.

Cú pháp:

```
int fclose(FILE *f)
```

Trong đó f là con trỏ tập tin được mở bởi hàm `fopen()`. Giá trị trả về của hàm là 0 báo rằng việc đóng tập tin thành công. Hàm trả về EOF nếu có xuất hiện lỗi. Ngoài ra, ta còn có thể sử dụng hàm `fcloseall()` để đóng tất cả các tập tin lại.

Cú pháp:

```
int fcloseall()
```

Kết quả trả về của hàm là tất cả các tập tin được đóng lại nếu thành công, ngược lại kết quả trả về là EOF.

1.2.4. Kiểm tra đến cuối tập tin

Cú pháp:

```
int feof(FILE *f)
```

Ý nghĩa: Kiểm tra xem đã chạm tới cuối tập tin hay chưa và trả về EOF nếu cuối tập tin được chạm tới, ngược lại trả về 0.

1.2.5. Di chuyển con trỏ tập tin về đầu tập tin

Khi ta đang thao tác một tập tin đang mở, con trỏ tập tin luôn di chuyển về phía cuối tập tin. Muốn cho con trỏ quay về đầu tập tin như khi mở nó, ta sử dụng hàm `rewind()`.

Cú pháp:

```
void rewind(FILE *f)
```

1.3. TRUY CẬP TẬP TIN VĂN BẢN

1.3.1. Ghi dữ liệu lên tập tin văn bản

1.3.1.1. Hàm putc()

Hàm này được dùng để ghi một ký tự lên một tập tin văn bản đang được mở để làm việc.

Cú pháp:

```
int putc(int c, FILE *f)
```

Trong đó, tham số c chứa mã Ascii của một ký tự nào đó. Mã này được ghi lên tập tin liên kết với con trỏ f. Hàm này trả về EOF nếu gặp lỗi.

1.3.1.2 Hàm fputs()

Hàm này dùng để ghi một chuỗi ký tự chứa trong vùng đệm lên tập tin văn bản.

Cú pháp:

```
int puts(const char *buffer, FILE *f)
```

Trong đó, buffer là con trỏ có kiểu char chỉ đến vị trí đầu tiên của chuỗi ký tự được ghi vào. Hàm này trả về giá trị 0 nếu buffer chứa chuỗi rỗng và trả về EOF nếu gặp lỗi.

1.3.1.3 Hàm fprintf()

Hàm này dùng để ghi dữ liệu có định dạng lên tập tin văn bản.

Cú pháp:

```
fprintf(FILE *f, const char *format, varexpr)
```

Trong đó:

format: chuỗi định dạng (giống với các định dạng của hàm printf()),

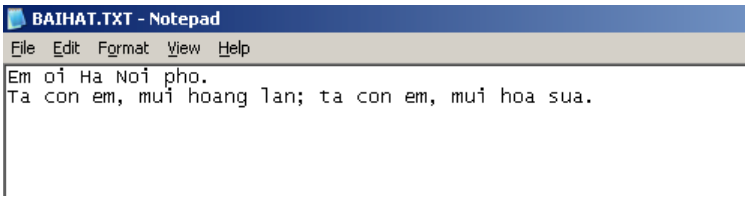
varexpr: danh sách các biểu thức, mỗi biểu thức cách nhau dấu phẩy (,).

Định dạng	Ý nghĩa
%d	Ghi số nguyên
%[.số chữ số thập phân] f	Ghi số thực có <số chữ số thập phân> theo quy tắc làm tròn số.
%o	Ghi số nguyên hệ bát phân
%x	Ghi số nguyên hệ thập lục phân
%c	Ghi một ký tự
%s	Ghi chuỗi ký tự
%e hoặc %E hoặc %g hoặc %G	Ghi số thực dạng khoa học (nhân 10 mũ x)

Ví dụ: Viết chương trình ghi chuỗi ký tự lên tập tin văn bản D:\\Baihat.txt

```
#include<stdio.h>
#include<conio.h>
int main()
{
 FILE *f; clrscr();
 f=fopen("D:\\Baihat.txt","r+");
 if (f!=NULL)
 {
     fputs("Em oi Ha Noi pho.\n",f);
     fputs("Ta con em, mui hoang lan; ta con em, mui hoa
sua.",f);
     fclose(f);
 }
 getch();
 return 0;
}
```

Nội dung tập tin Baihat.txt khi được mở bằng trình soạn thảo văn bản Notepad.



1.3.2. Đọc dữ liệu từ tập tin văn bản

1.3.2.1 Hàm getc()

Hàm này dùng để đọc dữ liệu từ tập tin văn bản đang được mở để làm việc.

Cú pháp:

```
int getc(FILE *f)
```

Hàm này trả về mã ASCII của một ký tự nào đó (kể cả EOF) trong tập tin liên kết với con trỏ f.

1.3.2.2 Hàm fgets()

Cú pháp:

```
char *fgets(char *buffer, int n, FILE *f)
```

Hàm này được dùng để đọc một chuỗi ký tự từ tập tin văn bản đang được mở ra và liên kết với con trỏ f cho đến khi đọc đủ n ký tự hoặc gặp ký tự xuống dòng '\n' (ký tự này cũng được đưa vào chuỗi kết quả) hay gặp ký tự kết thúc EOF (ký tự này không được đưa vào chuỗi kết quả).

Trong đó:

- buffer (vùng đệm): con trỏ có kiểu char chỉ đến cùng nhớ đủ lớn chứa các ký tự nhận được.
- n: giá trị nguyên chỉ độ dài lớn nhất của chuỗi ký tự nhận được.
- f: con trỏ liên kết với một tập tin nào đó.
- Ký tự NULL ('\0') tự động được thêm vào cuối chuỗi kết quả lưu trong vùng đệm.
- Hàm trả về địa chỉ đầu tiên của vùng đệm khi không gặp lỗi và chưa gặp ký tự kết thúc EOF. Ngược lại, hàm trả về giá trị NULL.

1.3.2.3 Hàm fscanf()

Hàm này dùng để đọc dữ liệu từ tập tin văn bản vào danh sách các biến theo định dạng.

Cú pháp:

```
fscanf(FILE *f, const char *format, varlist)
```

Trong đó:

- format: chuỗi định dạng (giống hàm scanf());
- varlist: danh sách các biến mỗi biến cách nhau dấu phẩy (,).

Ví dụ: Viết chương trình chép tập tin D:\Baihat.txt ở trên sang tập tin D:\Baica.txt.

```
#include<stdio.h>
#include<conio.h>
int main()
{
      FILE *f1,*f2;
      clrscr();
      f1=fopen("D:\\Baihat.txt","rt");
      f2=fopen("D:\\Baica.txt","wt");
      if (f1!=NULL && f2!=NULL)
      {
       int ch=fgetc(f1);
       while (! feof(f1))
       {
            fputc(ch,f2);
            ch=fgetc(f1);
       }
       fcloseall();
      }
      getch();
      return 0;
}
```

1.4. TRUY CẬP TẬP TIN NHỊ PHÂN

1.4.1 Ghi dữ liệu lên tập tin nhị phân - Hàm fwrite()

Cú pháp:

```
size_t fwrite(const void *ptr, size_t size, size_t n, FILE *f)
```

Trong đó:

- ptr: con trỏ chỉ đến vùng nhớ chứa thông tin cần ghi lên tập tin.
- n: số phần tử sẽ ghi lên tập tin.
- size: kích thước của mỗi phần tử.
- f: con trỏ tập tin đã được mở.
- Giá trị trả về của hàm này là số phần tử được ghi lên tập tin. Giá trị này bằng n trừ khi xuất hiện lỗi.

1.4.2 Đọc dữ liệu từ tập tin nhị phân - Hàm fread()

Cú pháp:

```
size_t fread(const void *ptr, size_t size, size_t n, FILE *f)
```

Trong đó:

- ptr: con trỏ chỉ đến vùng nhớ sẽ nhận dữ liệu từ tập tin.
- n: số phần tử được đọc từ tập tin.
- size: kích thước của mỗi phần tử.
- f: con trỏ tập tin đã được mở.
- Giá trị trả về của hàm này là số phần tử đã đọc được từ tập tin. Giá trị này bằng n hay nhỏ hơn n nếu đã chạm đến cuối tập tin hoặc có lỗi xuất hiện..

1.4.3 Di chuyển con trỏ tập tin - Hàm fseek()

Việc ghi hay đọc dữ liệu từ tập tin sẽ làm cho con trỏ tập tin dịch chuyển một số byte, đây chính là kích thước của kiểu dữ liệu của mỗi phần tử của tập tin. Khi đóng tập tin rồi mở lại nó, con trỏ luôn ở vị trí ngay đầu tập tin. Nhưng nếu ta sử dụng kiểu mở tập tin là "a" để ghi nối dữ liệu, con trỏ tập tin sẽ di chuyển đến vị trí cuối cùng của tập tin này. Ta cũng có thể điều khiển việc di chuyển con trỏ tập tin đến vị trí chỉ định bằng hàm fseek().

Cú pháp:

```
int fseek(FILE *f, long offset, int whence)
```

Trong đó:

f: con trỏ tập tin đang thao tác.

offset: số byte cần dịch chuyển con trỏ tập tin kể từ vị trí trước đó. Phần tử đầu tiên là vị trí 0.

whence: vị trí bắt đầu để tính offset, ta có thể chọn điểm xuất phát là:

0	SEEK_SET	Vị trí đầu tập tin
1	SEEK_CUR	Vị trí hiện tại của con trỏ tập tin
2	SEEK_END	Vị trí cuối tập tin

Kết quả trả về của hàm là 0 nếu việc di chuyển thành công. Nếu không thành công, 1 giá trị khác 0 (đó là 1 mã lỗi) được trả về.

1.4.4. Ví dụ

Ví dụ 1: Viết chương trình ghi lên tập tin CacSo.Dat 3 giá trị số (thực, nguyên, nguyên dài). Sau đó đọc các số từ tập tin vừa ghi và hiển thị lên màn hình.

```
#include<stdio.h>
#include<conio.h>
int main()
{
 FILE *f; clrscr();
 f=fopen("D:\\CacSo.txt","wb");
 if (f!=NULL)
 {
    double d=3.14;
    int i=101;
    long l=54321;
    fwrite(&d,sizeof(double),1,f);
    fwrite(&i,sizeof(int),1,f);
    fwrite(&l,sizeof(long),1,f);
    /* Doc tu tap tin*/
    rewind(f);
    fread(&d,sizeof(double),1,f);
    fread(&i,sizeof(int),1,f);
```

```
    fread(&l,sizeof(long),1,f);
    printf("Cac ket qua la: %f %d %ld",d,i,l);
    fclose(f);
}
 getch();
 return 0;
}
```

Ví dụ 2: Mỗi sinh viên cần quản lý ít nhất 2 thông tin: mã sinh viên và họ tên.

Viết chương trình cho phép lựa chọn các chức năng: nhập danh sách sinh viên từ bàn phím rồi ghi lên tập tin SinhVien.dat, đọc dữ liệu từ tập tin SinhVien.dat rồi hiển thị danh sách lên màn hình, tìm kiếm họ tên của một sinh viên nào đó dựa vào mã sinh viên nhập từ bàn phím.

Chúng ta nhận thấy rằng mỗi phần tử của tập tin SinhVien.Dat là một cấu trúc có 2 trường: mã và họ tên. Do đó, ta cần khai báo cấu trúc này và sử dụng các hàm đọc/ghi tập tin nhị phân với kích thước mỗi phần tử của tập tin là chính kích thước cấu trúc đó.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
typedef struct
{
    char Ma[10];
    char HoTen[40];
} SinhVien;
void WriteFile(char *FileName)
{
    FILE *f;
    int n,i;
    SinhVien sv;
    f=fopen(FileName,"ab");
    printf("Nhap bao nhieu sinh vien? ");scanf("%d",&n);
    fflush(stdin);
    for(i=1;i<=n;i++)
    {
     printf("Sinh vien thu %i\n",i);
     printf("- MSSV: ");gets(sv.Ma);
     printf("- Ho ten: ");gets(sv.HoTen);
     fwrite(&sv,sizeof(sv),1,f);
     fflush(stdin);
    }
    fclose(f);
    printf("Bam phim bat ky de tiep tuc");
    getch();
}
void ReadFile(char *FileName)
{
    FILE *f; SinhVien sv; f=fopen(FileName,"rb");
    printf("MSSV | Ho va ten\n");
    fread(&sv,sizeof(sv),1,f);
    while (!feof(f))
    {
     printf("%s | %s\n",sv.Ma,sv.HoTen);
```

```
        fread(&sv,sizeof(sv),1,f);
      }
      fclose(f);
      printf("Bam phim bat ky de tiep tuc!!!");
      getch();
}
void Search(char *FileName)
{
      char MSSV[10]; FILE *f;
      int Found=0; SinhVien sv;
      fflush(stdin);
      printf("Ma so sinh vien can tim: ");gets(MSSV);
      f=fopen(FileName,"rb");
      while (!feof(f) && Found==0)
      {
        fread(&sv,sizeof(sv),1,f);
        if (strcmp(sv.Ma,MSSV)==0) Found=1;
      }
      fclose(f);
      if (Found == 1)
        printf("Tim thay SV co ma %s. Ho ten la:
 %s",sv.Ma,sv.HoTen);
      else
        printf("Tim khong thay sinh vien co ma %s",MSSV);
      printf("\nBam phim bat ky de tiep tuc!!!");
      getch();
}
int main()
{
int c;
for (;;)
{
clrscr();
printf("1. Nhap DSSV\n");
printf("2. In DSSV\n");
printf("3. Tim kiem\n");
printf("4. Thoat\n");

printf("Ban chon 1, 2, 3, 4: "); scanf("%d",&c);
if(c==1)
WriteFile("d:\\SinhVien.Dat");
else if (c==2)
ReadFile("d:\\SinhVien.Dat");
else if (c==3) Search("d:\\SinhVien.Dat");
else break;
}
return 0;
}
```

Ngoài ra các chức năng trên, thư viện stdio.h còn định nghĩa một số hàm khác cho phép thao tác với tập tin, sinh viên có thể tham khảo trong phần trợ giúp.

❖ **Bài tập củng cố:**

1. Viết chương trình quản lý một tập tin văn bản theo các yêu cầu:

a- Nhập từ bàn phím nội dung một văn bản sau đó ghi vào đĩa.

b- Đọc từ đĩa nội dung văn bản vừa nhập và in lên màn hình.

c- Đọc từ đĩa nội dung văn bản vừa nhập, in nội dung đó lên màn hình và cho phép nối thêm thông tin vào cuối tập tin đó.

2. Viết chương trình cho phép thống kê số lần xuất hiện của các ký tự là chữ ('A'..'Z','a'..'z') trong một tập tin văn bản.

3. Viết chương trình đếm số từ và số dòng trong một tập tin văn bản.

4. Viết chương trình nhập từ bàn phím và ghi vào 1 tập tin tên là DMHH.DAT với mỗi phần tử của tập tin là 1 cấu trúc bao gồm các trường: Ma (mã hàng: char[5]), Ten (Tên hàng: char[20]). Kết thúc việc nhập bằng cách gõ ENTER vào Ma.

5. Viết chương trình cho phép nhập từ bàn phím và ghi vào 1 tập tin tên DSHH.Dat với mỗi phần tử của tập tin là một cấu trúc bao gồm các trường : mh (mã hàng: char[5]), sl (số lượng : int), dg (đơn giá: float), st (Số tiền: float) theo yêu cầu:

- Mỗi lần nhập một cấu trúc
- Trước tiên nhập mã hàng (mh), đưa mh so sánh với ma trong tập tin DMHH.DAT đã được tạo ra bởi bài tập 1, nếu mh=ma thì in tên hàng ngay bên cạnh mã hàng.
- Nhập số lượng (sl).
- Nhập đơn giá (dg).
- Tính số tiền = số lượng * đơn giá.
- Kết thúc việc nhập bằng cách gõ ENTER vào mã hàng và in danh sách ra màn hình theo mẫu như sau:

STT	MA HANG	TEN HANG	SO LG	DON GIA	SO TIEN
1	A0101	Sua Co gai Ha Lan	10	20000	200000
2	B0101	Sua Ong Tho	15	10000	150000

CHƯƠNG 2
SẮP THỨ TỰ NGOÀI

❖ **Mục tiêu học tập:** Sau khi học xong chương này, người học có thể vận dụng và cài đặt:
- Phương pháp trộn Run trên tập tin.
- Phương pháp trộn tự nhiên trên tập tin.
- Phương pháp trộn đa lối cân bằng trên tập tin.
- Phương pháp trộn đa pha trên tập tin.

Sắp thứ tự ngoài là sắp thứ tự trên tập tin. Khác với sắp xếp dãy trên bộ nhớ có số lượng phần tử nhỏ và truy xuất nhanh, tập tin có thể có số lượng phần tử rất lớn và thời gian truy xuất chậm. Do vậy việc sắp xếp trên các cấu trúc dữ liệu loại tập tin đòi hỏi phải áp dụng các phương pháp đặc biệt.

Chương này sẽ giới thiệu một số phương pháp như sau:

· Phương pháp trộn Run.

· Phương pháp trộn tự nhiên.

· Phương pháp trộn đa lối cân bằng(balanced multiway merging)

· Phương pháp trộn đa pha(PolyBước Merge)

2.1. PHƯƠNG PHÁP TRỘN RUN

Khái niệm cơ bản:

Run là một dãy liên tiếp các phần tử được sắp thứ tự.

Ví dụ: *1 2 3 4 5 là một Run gồm có 5 phần tử*

Chiều dài Run chính là số phần tử trong Run. Chẳng hạn, Run trong ví dụ trên có chiều dài là 5.

Như vậy, mỗi phần tử của dãy có thể xem như là 1 Run có chiều dài là 1. Hay nói khác đi, mỗi phần tử của dãy chính là một Run có chiều dài bằng 1.

Việc tạo ra một Run mới từ 2 Run ban đầu gọi là trộn Run (merge). Hiển nhiên, Run được tạo từ hai Run ban đầu là một dãy các phần tử đã được sắp thứ tự.

Giải thuật:

Giải thuật sắp xếp tập tin bằng phương pháp trộn Run có thể tóm lược như sau:

Input: f0 là tập tin cần sắp thứ tự.

Output: f0 là tập tin đã được sắp thứ tự.

Gọi f1, f2 là 2 tập tin trộn.

Các tập tin f0, f1, f2 có thể là các tập tin tuần tự (text file) hay có thể là các tập tin truy xuất ngẫu nhiên (File of <kiểu>)

Bước 1:

- Giả sử các phần tử trên f0 là:

  24 12 67 33 58 42 11 34 29 31
- f1 ban đầu rỗng, và f2 ban đầu cũng rỗng.
- Thực hiện phân bố m=1 phần tử lần lượt từ f0 vào f1 và f2:

  f1: 24 67 58 11 29

  f2: 12 33 42 34 31
- Trộn f1, f2 thành f0:

  f0: 12 24 33 67 42 58 11 34 29 31

Bước 2:

- Phân bố m=2 phần tử lần lượt từ f0 vào f1 và f2:
    f1: 12 24 42 58 29 31
    f2: 33 67 11 34
- Trộn f1, f2 thành f0:
    f0: 12 24 33 67 11 34 42 58 29 31

Bước 3:

- Tương tự bước 2, phân bố m=4 phần tử lần lượt từ f0 vào f1 và f2, kết quả thu được như sau:
    f1: 12 24 33 67 29 31
    f2: 11 34 42 58
- Trộn f1, f2 thành f0:
    f0: 11 12 24 33 34 42 58 67 29 31

Bước 4:

- Phân bố m=8 phần tử lần lượt từ f0 vào f1 và f2:
    f1: 11 12 24 33 34 42 58 67
    f2: 29 31
- Trộn f1, f2 thành f0:
    f0: 11 12 24 29 31 33 34 42 58 67

Bước 5:

Lặp lại tương tự các bước trên, cho đến khi chiều dài m của Run cần phân bổ lớn hơn chiều dài n của f0 thì dừng. Lúc này f0 đã được sắp thứ tự xong.

Cài đặt:

```
#include "stdio.h"
#include "conio.h"
int p,n;
void tao_file(void)
{
     //Tao  file co n phan tu
     int i,x;
     FILE *fp;
     fp=fopen("D:\\Bang.txt","wb");
     printf("Cho biet so phan tu : ");
     scanf("%d", &n);
     for(i=0;i<n;i++)
     {
          printf("Nhap so thu %d : ",i+1);
          scanf("%d", &x);
          fprintf(fp,"%3d",x);
     }
     fclose(fp);
}

void xuat_file(void)
{
     //Hien thi noi dung file len man hinh

     int x;
     FILE *fp;
     fp=fopen("D:\\Bang.txt","rb");
     int i=0;
     while(i<n)
```

```
          {
               fscanf(fp,"%d",&x);
               printf("%3d",x);
               i++;
          }
          fclose(fp);
     }

     void chia(FILE *a, FILE *b, FILE *c, int p)
     {
          //Chia xoay vong file a cho file b va file c moi lan
p phan tu cho den khi het file a.

          int dem, x;
          a=fopen("D:\\Bang.txt","rb");
          b=fopen("D:\\Bang1.txt","wb");
          c=fopen("D:\\Bang2.txt","wb");
          while(!feof(a))
          {
               //Chia p phan tu cho b

               dem=0;
               while((dem<p) && (!feof(a)))
               {
                    fscanf(a,"%3d",&x);
                    fprintf(b,"%3d",x);
                    dem++;
               }

               //Chia p phan tu cho c

               dem=0;
               while((dem<p) && (!feof(a)))
               {
                    fscanf(a,"%3d",&x);
                    fprintf(c,"%3d",x);
                    dem++;
               }
          }
          fclose(a);
          fclose(b);
          fclose(c);
     }

     void tron(FILE *b, FILE *c, FILE *a, int p)
     {
          //Tron p phan tu tren b voi p phan tu tren c thanh
2*p phan tu tren a cho den khi file b hoac c het

          int stop, x, y, l, r;
          a=fopen("D:\\Bang.txt","wb");
```

```
            b=fopen("D:\\Bang1.txt","rb");
            c=fopen("D:\\Bang2.txt","rb");
            while((!feof(b)) && (!feof(c)))
            {
                  l=0; //so phan tu cua b da ghi het len a
                  r=0; //so phan tu cua c da ghi het len a
                  fscanf(b,"%3d",&x);
                  fscanf(c,"%3d",&y);
                  stop=0;
                  while((l!=p) && (r!=p) && (!stop))
                  {
                        if(x<y)
                        {
                              fprintf(a,"%3d",x);
                              l++;
                              if((l<p) && (!feof(b)))
                                    fscanf(b,"%3d",&x);  //chua  du  p
phan tu va chua het file b
                              else
                              {
                                    fprintf(a,"%3d",y);
                                    r++;
                                    if((feof(b)))
                                          stop=1;
                              }
                        }
                        else
                        {
                              fprintf(a,"%3d",y);
                              r++;
                              if((r<p) && (!feof(c)))
                                    fscanf(c,"%3d",&y);  //chua  du  p
phan tu va chua het file c
                              else
                              {
                                    fprintf(a,"%3d",x);
                                    l++;
                                    if((feof(c)))
                                          stop=1;
                              }
                        }
                  }

                  //chep phan tu con lai cua p phan tu tren b len
a

                  while((!feof(b)) && (l<p))
                  {
                        fscanf(b,"%3d",&x);
                        fprintf(a,"%3d",x);
                        l++;
```

```
                }

                //chep phan tu con lai cua p phan tu tren c len
a

                while((!feof(c)) && (r<p))
                {
                        fscanf(c,"%3d",&y);
                        fprintf(a,"%3d",y);
                        r++;
                }
        }
        if(!feof(b))
        {
                //chep phan tu con lai cua b len a
                while(!feof(b))
                {
                        fscanf(b,"%3d",&x);
                        fprintf(a,"%3d",x);
                }
        }
        if(!feof(c))
        {
                //chep phan tu con lai cua c len a
                while(!feof(c))
                {
                        fscanf(c,"%3d",&x);
                        fprintf(a,"%3d",x);
                }
        }
        fclose(a);
        fclose(b);
        fclose(c);
}

void main(void)
{
        FILE *a, *b, *c;
        tao_file();
        xuat_file();
        p=1;
        while(p<n)
        {
                chia(a,b,c,p);
                tron(b,c,a,p);
                p=2*p;
        }
        printf("\n");
        xuat_file();
        getch();
}
```

2.2. PHƯƠNG PHÁP TRỘN TỰ NHIÊN

Giải thuật:

Trong phương pháp trộn đã trình bày ở trên, giải thuật không tận dụng được chiều dài cực đại của các Run trước khi phân bổ; do vậy, việc tối ưu thuật toán chưa được tận dụng.

Đặc điểm cơ bản của phương pháp trộn tự nhiên là tận dụng độ dài "tự nhiên" của các Run ban đầu; nghĩa là, thực hiện việc trộn các Run có độ dài cực đại với nhau cho đến khi dãy chỉ bao gồm một Run: dãy đã được sắp thứ tự.

Input: f0 là tập tin cần sắp thứ tự.

Output: f0 là tập tin đã được sắp thứ tự.

Lặp Cho đến khi dãy cần sắp chỉ gồm duy nhất một Run.

Phân bố:

- Chép một Run có thứ tự vào tập tin phụ fi (i>=1). Khi chấm dứt Run này, biến Eor (End of Run) có giá trị True.
- Chép Run con có thứ tự kế tiếp vào tập tin phụ kế tiếp fi+1 (xoay vòng).
- Việc phân bố kết thúc khi kết thúc tập tin cần sắp f0.

Trộn:

- Trộn 1 Run trong f1 và1 Run trong f2 vào f0.
- Việc trộn kết thúc khi duyệt hết f1 và hết f2 (hay nói cách khác, việc trộn kết thúc khi đã có đủ n phần tử cần chép vào f0).

Cài đặt:

```
#include "stdio.h"
#include "conio.h"
#include "stdlib.h"
#include "iostream.h"

void CreateFile(FILE *Ft, int);
void ListFile(FILE *);
void Distribute();
void Copy(FILE *, FILE *);
void CopyRun(FILE *, FILE *);
void MergeRun();
void Merge();

typedef int DataType;
FILE *F0, *F1, *F2;
int M,N,Eor;
//Bien Eor dung de kiem tra ket thuc Run hoac File
DataType X1, X2, X, Y;

void CreateFile(FILE *Ft, int Num)
{
     //Tao file co ngau nhien n phan tu
     randomize();
     Ft=fopen("D:\\Bang.txt","wb");
     for(int i=0;i<Num;i++)
     {
          X=random(30);
          fprintf(Ft,"%3d",X);
     }
     fclose(Ft);
}
```

```
void ListFile(FILE *Ft)
{
     //Hien thi noi dung cua file len man hinh
     DataType X, I=0;
     Ft=fopen("D:\\Bang.txt","rb");
     while(I<N)
     {
          fscanf(Ft,"%3d",&X);
          cout<<""<<X;
          I++;
     }
     printf("\n\n");
     fclose(Ft);
}

void Copy(FILE *Fi, FILE *Fj)
{
     //Doc phan tu X tu tap tin Fi, ghi X vao Fj
     //Eor == 1, Neu het Run(tren Fi) hoac het File Fj

     fscanf(Fi,"%3d",&X);
     fprintf(Fj,"%3d",X);
     if(!feof(Fi))
     {
          fscanf(Fi,"%3d",&Y);
          long curpos=ftell(Fi)-2;
          fseek(Fi,curpos,SEEK_SET);
     }
     if(feof(Fi))
          Eor=1;
     else
          Eor=(X>Y)?1:0;
}

void CopyRun(FILE *Fi, FILE *Fj)
{
     //Chep 1 Run tu File Fi vao File Fj
     do
     {
          Copy(Fi, Fj);
     }
     while(!Eor);
}

void Distribute()
{
     //Phan bo luan phien cac Run tu nhien tu F0 vao F1 va
F2

     do
```

```
        {
            CopyRun(F0, F1);
            if(!feof(F0))
                CopyRun(F0, F2);
        }
        while(!feof(F0));
        fclose(F0);
        fclose(F1);
        fclose(F2);
    }

    void MergeRun()
    {
        do
        {
            fscanf(F1,"%3d",&X1);
            long curpos=ftell(F1)-2;
            fseek(F1, curpos, SEEK_SET);
            fscanf(F2, "%3d",&X2);
            curpos=ftell(F2)-2;
            fseek(F2,curpos, SEEK_SET);
            if(X1<=X2)
            {
                Copy(F1, F0);
                if(Eor)
                    CopyRun(F2, F0);
            }
            else
            {
                Copy(F2, F0);
                if(Eor)
                    CopyRun(F1, F0);
            }
        }
        while(!Eor);
    }

    void Merge()
    {
        //tron cac Run tu F1 va F2 vao F0
        while((!feof(F1)) && (!feof(F2)))
        {
            MergeRun();
            M++;
        }
        while(!feof(F1))
        {
            CopyRun(F1, F0);
            M++;
        }
        while(!feof(F2))
```

```
        {
            CopyRun(F2, F0);
            M++;
        }
        fclose(F0);
        fclose(F1);
        fclose(F2);
}

void main()
{
        clrscr();
        randomize();
        cout<<"Nhap so phan tu : ";
        cin>>N;
        CreateFile(F0,N);
        ListFile(F0);
        do
        {
            F0=fopen("D:\\Bang.txt","rb");
            F1=fopen("D:\\Bang1.txt","wb");
            F2=fopen("D:\\Bang2.txt","wb");
            Distribute();
            F0=fopen("D:\\Bang.txt","wb");
            F1=fopen("D:\\Bang1.txt","rb");
            F2=fopen("D:\\Bang2.txt","rb");
            M=0;
            Merge();
        }while(M!=1);
        ListFile(F0);
        getch();
}
```

2.3. PHƯƠNG PHÁP TRỘN ĐA LỐI CÂN BẰNG

(Balanced MultiWay Merging)

Giải thuật:

Các phương pháp đã trình bày ở trên chủ yếu dựa trên hai thao tác: *phân phối* và *trộn* các Run. Thời gian thực thi các phương pháp này chủ yếu bị chi phối bởi thời gian phân phối các Run từ tập tin f0 vào các tập tin phụ f1 và f2.

Phương pháp trộn đa lối cân bằng sẽ khắc phục được nhược điểm này.

Ý tưởng cơ bản của phương pháp trộn đa lối cân bằng là sử dụng N chẵn tập tin.

Input: f0 là tập tin cần sắp thứ tự.

Output: f0 là tập tin đã được sắp thứ tự.

Bước 0: Đặt nh= N/2

Bước 1:

- Phân phối các Run luân phiên vào f[1], f[2], .. f[nh]

Bước 2:

- Lặp lại bước 3 Cho đến khi dãy sau khi trộn chỉ gồm duy nhất một Run

Bước 3:

- Trộn các Run của f[1] .. f[nh] và luân phiên phân phối vào các tập tin f[nh+1] .. f[n].
- Nếu số Run q sau khi trộn > 1 thì trộn các Run từ f[nh+1] .. f[n] vào f[1].. f[nh].

Ngược lại: kết thúc giải thuật

Ghi chú

T : lưu trữ chỉ số tập tin trộn và phân phối.

Các tập tin f[T[1]].. f[T[nh]] sẽ trộn vào các tập tin f[T[nh+1]] .. f[T[n]].

Ta: lưu trữ chỉ số tập tin đang được trộn

Cài đặt:

```
#include"stdio.h"
#include"conio.h"
#include"stdlib.h"
#include"string.h"
#define n 4
/**/
int   Copy_Run(FILE   **f,FILE   **fx,int   ele_start,int
&ele_new_Run)
{
     int cur=ele_start,old,Eof;
     do
     {
          fwrite(&cur,sizeof(cur),1,*fx);
          old=cur;
          Eof=fread(&cur,sizeof(cur),1,*f);
          if(Eof<=0)
          {
               ele_new_Run=NULL;
               return -1;// het file
          }
     }
     while(old<=cur);
     ele_new_Run=cur;
     return 0;
}

void Distribute_Run(char *fa,char *fax[],int &q)
{
     int current,old,Eof,new_Run=0,tx;
     int i=0;
     FILE *f,*fx[15];
     f=fopen(fa,"w+");
     for(i=0;i<n;i++)
     fx[i]=fopen(fax[i],"w+");
     Eof=fread(&current,sizeof(current),1,f);
     if(Eof<=0) return;
     do
     {
          Eof=Copy_Run(&f,&fx[i],current,new_Run);
          current=new_Run;
          i=i%n+1;
          q++;
     } while(Eof>0);
}
/**/
```

```
void Distribute(char *fa,char *fax[],int &q)
{
     FILE *f,*fx[7];
     f=fopen(fa,"w+");
     int j;
     for(int i=0;i<n;i++)
     {
          remove(fax[i]);
          fx[i]=fopen(fax[i],"w+");
     }
     j=n;
     q=0;
     int current,old;
     do
     {
          if(j<n-1) j++;
          else j=0;
          q++;
          fread(&current,sizeof(current),1,f);
          do
          {
               old=current;
               fwrite(&current,sizeof(current), 1, fx[j]);
               fread(&current,sizeof(current),1,f);
          } while(!feof(f)&&old<current);
     } while(!feof(f));
}
/**/
void Merge(char *fa[],int &q)
{
     FILE *f[20];
     int i,j,k1,k2,min,mx,Eof,x,tx;
     int t[20],ta[20];
     int current[100],cur;
     for(i=0;i<2*n;i++)
     {
          t[i]=i;
          f[i]=fopen(fa[i],"w+");
     }

     do
     {
          if(q<n) k1=q;
          else k1=n;
          for(i=0;i<k1;i++)
          {
               fread(&current[i],sizeof(current[i]),    1,
          f[t[i]]);
               ta[i]=t[i];
          }
          q=0;
```

```
j=n;

do
{
      k2=k1;
      q++;
      do
      {
            i=0;mx=0;
            min=current[i];
            while(i<k2-1)
            {
                  i++;
                  x=current[i];
                  if(x<min)
                  {
                        min=x;
                        mx=i;
                  }
            }
            fwrite(&min,sizeof(min),1,f[t[j]]);
            Eof=fread(&cur,     sizeof(cur),     1,
      f[ta[mx]]);
            if(Eof<=0)
            {
                  remove(fa[ta[mx]]);
                  ta[mx]=ta[k2];
                  ta[k2]=ta[k1];
                  k1=k1-1;
                  k2--;
            }
            else
            {
                  if(min>cur)
                  {
                        tx=ta[mx];
                        ta[mx]=ta[k2];
                        ta[k2]=tx;
                        k2--;
                  }
            }
      } while(k2!=0);
      if(j<n*2-1)
            j++;
      else j=n;
} while(k1!=0);
      for(i=0;i<n;i++)
      {
            tx=t[i];
            t[i]=ta[n+i];
            t[n+i]=tx;
```

```
                }
        } while(q!=1);
        fcloseall();
}
/**/
void Copy(char *fa,char *ga)
{
        int current;
        FILE *f,*g;
        f=fopen(fa,"w+");
        g=fopen(ga,"w+");
        while(!feof(f))
        {
                fread(&current,sizeof(current),1,f);
                fwrite(&current,sizeof(current),1,g);
        }
        fcloseall();
}

/**/
void Taofile(char *filename,int k)
{
        randomize();
        int t;
        FILE *f;
        f=fopen(filename,"w+");
        for(int i=0;i<k;i++)
        {
                t=random(1000);
                fwrite(&t,sizeof(t),1,f);
        }
        fcloseall();
}

void xuat(char *filename)
{
        int cur;
        FILE *f;
        f=fopen(filename,"rb");
        while(fread(&cur,sizeof(cur),1,f)>0)
                printf("%5d",cur);
        fcloseall();
}
/**/
void main()
{
        char *filename[]= {"e:\\f1.txt", "e:\\f1.txt",
"e:\\f1.txt","e:\\f1.txt", "e:\\f1.txt", "e:\\f1.txt",
"e:\\f1.txt", "e:\\f1.txt", "e:\\f1.txt", "e:\\f1.txt"};
        char*f="e:\\tRung.txt";
        clrscr();
```

```
        Taofile(f,20);
        int q;
        xuat(f);
        printf("\nfile f sau khi xap xep\n");
        for(int i=0;i<n;i++)
        Distribute(f,filename,q);
        Merge(filename,q);
        Copy(f,filename[0]);
        xuat(f);
        getch();
}
```

2.4. PHƯƠNG PHÁP TRỘN ĐA PHA

(PolyBước Merge)

Phương pháp trộn đa lối cân bằng đã loại bỏ các phép sao chép thuần túy thông qua việc gộp quá trình phân phối và quá trình trộn trong cùng một giai đoạn. Tuy nhiên các tập tin các tập tin chưa được sử dụng một cách có hiệu quả bởi vì trong cùng một lần duyệt thì phân nửa số tập tin luôn luôn giữ vai trò trộn (nguồn) và phân nửa số tập tin luôn luôn giữ vai trò phân phối (đích). Ta có thể cải tiến phương pháp trên bằng cách giải quyết thay đổi vai trò của các tập tin trong cùng một lần duyệt phương pháp này gọi là phương pháp trộn đa pha.

Ta xét ví dụ sau với 3 tập tin f_1, f_2, f_3

Bước 1: Phân phối luân phiên các Run ban đầu của f_0 vào f_1 và f_2

Bước 2: Trộn các Run của f_1, f_2 vào f_3 . Giải thuật kết thúc nếu f_3 chỉ có một Run

Bước 3: Chép nửa Run của f_3 vào f_1

Bước 4: Trộn các Run của f_1 và f_3 vào f_2. Giải thuật kết thúc nếu f_2 chỉ có một Run.

Bước 5: Chép nửa số Run của f_2 vào f_1. Lặp lại *bước 2.*

Phương pháp này còn có nhược điểm là mất thời gian sao chép nửa số Run của tập tin này vào tập tin kia. Việc sao chép này có thể loại bỏ nếu ta bắt đầu với F_n Run của tập tin f_1 và F_{n-1} Run của tập tin f_2, với F_n và F_{n-1} là các số liên tiếp trong dãy Fibonacci.

Chúng ta xem xét các ví dụ sau:

Ví dụ 1: *Trường hợp n=7, tổng số Run ban đầu là 13+8=21 Run*

Bước	F 1	F2	F3	
0	1, 1, 1, 1, 1, 1, 1, 1	1, 1, 1, 1, 1		Sắp xếp
1	1, 1, 1,		2, 2, 2, 2, 2	Trộn 1
2		3, 3, 3	2, 2	Trộn 2
3	5, 5	3		Trộn 3
4	5		8	Trộn 4
5		13		Trộn 5
6	21			Trộn 6

Bước 0: Phân phối các Run ban đầu

Bước 1: Trộn 8 Run của f1 và f2 vào **f3**

Bước 2: Trộn 5 Run của f1 và f3 vào **f2**

Bước 3: Trộn 3 Run của f2 và f3 vào **f1**

Bước 4: Trộn 2 Run của f1 và f2 vào **f3**
Bước 5: Trộn 1 Run của f1 và f3 vào **f2**
Bước 6: Trộn 1 Run của f2 và f3 vào **f1**

Ví dụ 2:

Bước	T6	T5	T4	T3	T2	T1	Tổng số Runs được xữ lý
Bước 0	1^{31}	1^{30}	1^{28}	1^{24}	1^{16}	-	129
Bước 1	1^{15}	1^{14}	1^{12}	1^{8}	-	5^{16}	80
Bước 2	1^{7}	1^{6}	1^{4}	-	9^{8}	5^{8}	72
Bước 3	1^{3}	1^{2}	-	17^{4}	9^{4}	5^{4}	68
Bước 4	1^{1}	-	33^{2}	17^{2}	9^{2}	5^{2}	66
Bước 5	-	65^{1}	33^{1}	17^{1}	9^{1}	5^{1}	65
Bước 6	129^{1}	-	-	-	-	-	129

Bước 0: Phân phối các Run ban đầu
Bước 1: Trộn 16 Run từ T2 đến T6 vào **T1**
Bước 2: Trộn 8 Run của T1, T3, T4, T5, T6 vào **T2**
Bước 3: Trộn 4 Run của T1, T2, T4, T5, T6 vào **T3**
Bước 4: Trộn 2 Run của T6, T5, T3, T1, T6 vào **T4**
Bước 5: Trộn 1 Run của T1, T2, T3, T4, T6 vào **T5**
Bước 6: Trộn 1 Run từ T1 đến T5 vào **T6.**

Xem xét bảng trên (từ dưới lên) chúng ta thấy có 7 bước phân bố theo dãy Fibonacci bậc 4 là: {1,0,0,0,0}, {1,1,1,1,1}, {2,2,2,2,1}, {4,4,4,3,2}, {8,8,7,6,4}, {16,15,14,12,8}, và {31,30,28,24,16}.

Với Số tập tin T=6 bảng sau cho thấy số Run ban đầu được phân bố thích hợp:

Phân bố Fibonacci hoàn hảo với T=6						
Mức	**T6**	**T5**	**T4**	**T3**	**T2**	**Tổng số Runs được xữ lý**
0	1	0	0	0	0	1
1	1	1	1	1	1	5
2	2	2	2	2	1	9
3	4	4	4	3	2	17
4	8	8	7	6	4	33
5	16	15	14	12	8	65
6	31	30	28	24	16	129
7	61	59	55	47	31	253
8	120	116	108	92	61	497

-	-	-	-	-	-	-
n	an	bn	cn	dn	en	tn
n+1	an+bn	an+cn	an+dn	an+en	an	tn+4an

Trong ví dụ 1, số Run phan phối ban đầu cho các tập tin là 2 số Fibonacci kế tiếp nhau của dãy Fibonacci bậc 1: 0, 1, 1, 2, 3, 5, 8 . . .

Trong ví dụ 2 số Run ban đầu phân bố cho các tập tin theo dãy Fibonacci bậc 4: 0, 0, 0, 0, 1, 1, 2, 4, 8, 16 . . .

Dãy Fibonacci bậc p tổng quát được định nghĩa như sau:

$F^{(p)}_n = F^{(p)}_{n-1} + ... + F^{(p)}_{n-2} + ... + F^{(p)}_{n-p}$, *với* $n>=p$

Và $F^{(p)}_n = 0$, *với* $0 <= n <= p-2$; $F^{(p)}_{p-1} = 1$

Dãy Fibonacci thông thường là dãy Fibonacci bậc 1.

Thông thường nếu chúng ta lấy p= T-1, phân bố trộn đa pha đối với p tập tin sẽ tương ứng với số Fibonacci bậc p.

❖ **Bài tập củng cố:**

1. Vận dụng các bước sắp xếp của từng phương pháp trộn Run, tự nhiên, đa lối cân bằng và đa pha trên tập tin f0:

27 9 17 35 16 32 11 22 25 6 33 1 19 3 12 28 14 8 20

30 23 4

2. Cài đặt bốn phương pháp trộn với tập tin f0 là tập tin dạng văn bản.

CHƯƠNG 3
BẢNG BĂM (HASH TABLE)

❖ **Mục tiêu học tập:** Sau khi học xong bài này, người học có thể:
- Hiểu và vận dụng được hàm băm và các phép toán trên bảng băm.
- Vận dụng và cài đặt các phương pháp băm: băm nối kết, băm tuyến tính và băm kép.

Hàm băm được đề xuất và hiện thực trên máy tính từ những năm 50 của thế kỷ 20. Nó dựa trên ý tưởng: chuyển đổi khóa thành một số (xử lý băm) và sử dụng số này để đánh chỉ số cho bảng dữ liệu.

Các phép toán trên các cấu trúc dữ liệu như danh sách, cây nhị phân,… phần lớn được thực hiện bằng cách so sánh các phần tử của cấu trúc, do vậy thời gian truy xuất không nhanh và phụ thuộc vào kích thước của cấu trúc. Chương này sẽ khảo sát một cấu trúc dữ liệu mới được gọi là bảng băm(hash table). Các phép toán trên bảng băm sẽ giúp hạn chế số lần so sánh, và vì vậy sẽ cố gắng giảm thiểu được thời gian truy xuất. Độ phức tạp của các phép toán trên bảng băm thường có bậc là 0(1) và không phụ thuộc vào kích thước của bảng băm.

Chương này sẽ giới thiệu các chủ đề và các phép toán chính thường dùng trên cấu trúc bảng băm:

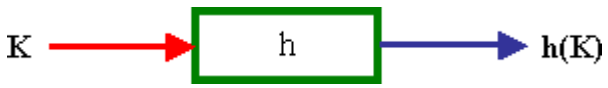
· Phép băm hay hàm băm (hash function)
· Tập khoá của các phần tử trên bảng băm
· Tập địa chỉ trên bảng băm
· Thêm phần tử vào bảng băm
· Xoá một phần tử trên bảng băm
· Tìm kiếm trên bảng băm

Thông thường bảng băm được sử dụng khi cần giải quyết những bài toán có các cấu trúc dữ liệu lớn và được lưu trữ ở bộ nhớ ngoài.

3.1. HÀM BĂM (Hash Function)

Định nghĩa:

Trong hầu hết các ứng dụng, khoá được dùng như một phương thức để truy xuất dữ liệu một cách gián tiếp. Hàm được dùng để ánh xạ một khoá vào một dãy các số nguyên và dùng các giá trị nguyên này để truy xuất dữ liệu được gọi là hàm băm (hình 1)



Hình 3.1

Như vậy, hàm băm là hàm biến đổi khóa của phần tử thành địa chỉ trên bảng băm.

Khóa có thể là dạng số hay số dạng chuỗi.

Hàm băm tốt thỏa mãn các điều kiện sau:
- Tính toán nhanh.
- Các khoá được phân bố đều trong bảng.
- Ít xảy ra đụng độ.

Giải quyết vấn đề băm với các khoá không phải là số nguyên:
- Tìm cách biến đổ khoá thành số nguyên

- Ví dụ loại bỏ dấu '-' trong mã số 9635-8904 đưa về số nguyên 96358904
- Đối với chuỗi, sử dụng giá trị các ký tự trong bảng mã ASCII
- Sau đó sử dụng các hàm băm chuẩn trên số nguyên.

Hàm Băm sử dụng Phương pháp chia

· Dùng số dư:
- $h(k) = k \bmod m$
- k là khoá, m là kích thước của bảng.

· vấn đề chọn giá trị m

· $m = 2^n$ (không tốt)

· nếu chọn m= 2^n thông thường không tốt h(k) = k mod 2^n sẽ chọn cùng n bits cuối của k

· m là nguyên tố (tốt)

· Gia tăng sự phân bố đều

· Thông thường m được chọn là số nguyên tố gần với 2^n
- Chẳng hạn bảng ~4000 mục, chọn m = 4093

Hàm Băm sử dụng Phương pháp nhân

· Sử dụng
- $h(k)$ = m ($k\,A$ mod 1)
- k là khóa, m là kích thước bảng, A là hằng số: $0 < A < 1$

· *Chọn m và A*
- M thường chọn m = 2p
- Sự tối ưu trong việc chọn A phụ thuộc vào đặc trưng của dữ liệu, theo Knuth chọn A = 0.618033987 được xem là tốt.

Phép băm phổ quát

· Việc chọn hàm băm không tốt có thể dẫn đến xác suất đụng độ lớn.

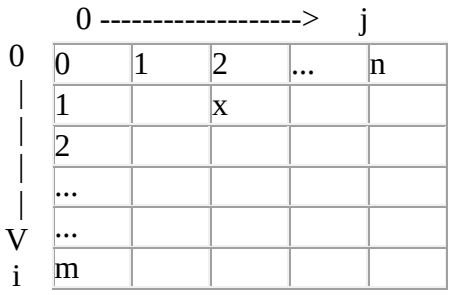
· Giải pháp:
- Lựa chọn hàm băm h ngẫu nhiên.
- Chọn hàm băm độc lập với khóa.
- Khởi tạo một tập các hàm băm H phổ quát và từ đó H được chọn ngẫu nhiên.

Một cách tổng quát, với một hàm băm, nhiều khoá khác nhau có thể cho cùng một giá trị băm. Trong tình huống này xảy ra sự xung đột (collision) và cần thiết phải giải quyết sự đụng độ này. Một trong những phương pháp giải quyết sự xung đột với thời gian nhanh là sử dụng các cấu trúc danh sách đặc, hay danh sách kề có kích thước cố định. (xem mục 3.4)

Các cấu trúc bảng băm đơn giản, thường được cài đặt bằng các danh sách kề. Do vậy, để truy xuất một phần tử trên các bảng băm thuộc loại này, chỉ cần hai khóa tương ứng với hàng thứ i và cột thứ j để định vị một phần tử trên bảng.

Bảng băm chữ nhật (m hàng, n cột):

Mỗi phần tử trên bảng chữ nhật tương ứng với hai khóa tương ứng hàng thứ i và cột thứ j, địa chỉ phần tử này trên danh sách kề được xác định qua hàm băm:



Hình 3.2. Bảng băm chữ nhật

0	1	2	3	...	n-1	n	n+1	n+2	...	m x n

Danh sách kề mô tả bảng băm hình chữ nhật

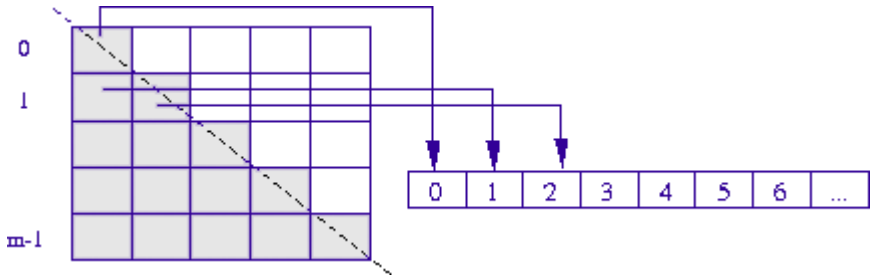
Bảng băm: phần tử x thuộc hàng 2 cột 3 - f(1,2) = n + 3

Tổng quát, phần tử thuộc hàng i, cột j được cho bởi công thức:

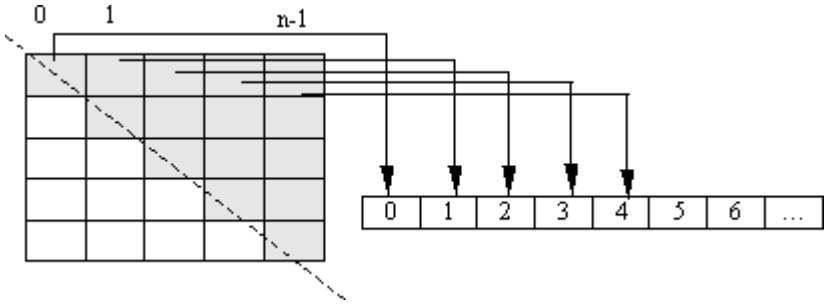
f(i,j) =n_i + j (n là số cột của bảng chữ nhật)

Bảng băm tam giác dưới (m hàng) và bảng băm tam giác trên (n cột):

Hình sau là bảng tam giác dưới m hàng



Hình 3.3.a Bảng băm tam giác dưới m hàng



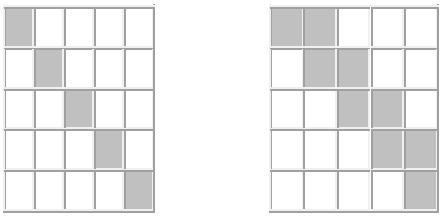
Hình 3.3.b Bảng băm tam giác trên n cột

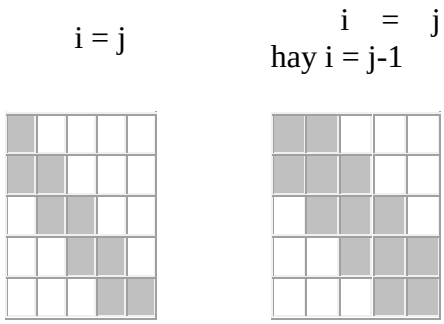
Mỗi phần tử trên bảng tam giác dưới (hình 3.3.a) tương ứng với hai khóa hàng i, cột j(i>=j), địa chỉ phần tử này trên danh sách kề được xác định qua hàm băm:

f(i,j)=i(i+1)/2 + j

Bảng băm đường chéo (n cột):

Hình sau là các dạng bảng đường chéo n cột, hãy xác định hàm băm cho các bảng đường chéo này.





Hình 3.4. Các bảng băm đường chéo

Như đã giới thiệu ở phần trên, với mỗi bảng băm đơn giản chúng ta cần xây dựng một hàm băm để truy xuất dữ liệu lưu trữ trong các phần tử trên bảng băm. Hàm băm thường có dạng công thức tổng quát **HF(key)** hay **f(khoá)** hoặc được tổ chức ở dạng bảng tra gọi là bảng truy xuất (access table).

3.2. BẢNG BĂM (Hash Table - ADT)

Phần này sẽ trình bày các vấn đề chính:

- Mô tả cấu trúc bảng băm tổng quát (thông qua hàm băm, tập khóa, tập địa chỉ…)
- Các phép toán trên bảng băm như thêm phần tử (insert), loại bỏ (remove), tìm kiếm (search), …

Bảng băm :

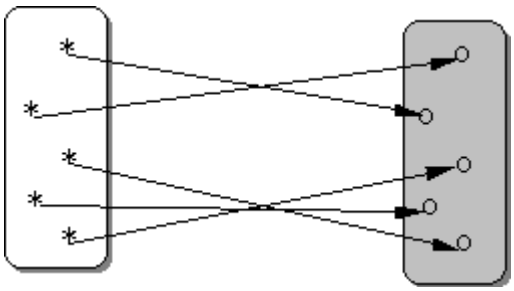
a. Mô tả dữ liệu

Giả sử

· K: tập các khoá (set of keys)

· M: tập các địa chỉ (set of addresses).

· HF(k): hàm băm dùng để ánh xạ một khoá k từ tập các khoá K thành một địa chỉ tương ứng trong tập M.



Hình 3.5. Hàm băm

b. Các phép toán trên bảng băm

· Khởi tạo (*Initialize*): Khởi tạo bảng băm, cấp phát vùng nhớ hay qui định số phần tử (kích thước) của bảng băm

· Kiểm tra rỗng (*Empty*): kiểm tra bảng băm có rỗng hay không?

· Lấy kích thước của bảng băm (*Size*): Cho biết số phần tử hiện có trong bảng băm

· Tìm kiếm (*Search*): Tìm kiếm một phần tử trong bảng băm theo khoá k chỉ định trước.

· Thêm mới phần tử (*Insert*): Thêm một phần tử vào bảng băm. Sau khi thêm số phần tử hiện có của bảng băm tăng thêm một đơn vị.

· Loại bỏ (*Remove*): Loại bỏ một phần tử ra khỏi bảng băm, và số phần tử sẽ giảm đi một.

· Sao chép (*Copy*): Tạo một bảng băm mới từ một bảng băm cũ đã có.

· Duyệt (*Traverse*): duyệt bảng băm theo thứ tự địa chỉ từ nhỏ đến lớn.

Các Bảng băm thông dụng:

Với mỗi loại bảng băm cần thiết phải xác định tập khóa K, xác định tập địa chỉ M và xây dựng hàm băm HF cho phù hợp.

Mặt khác, khi xây dựng hàm băm cũng cần thiết phải tìm kiếm các giải pháp để giải quyết sự xung đột, nghĩa là giảm thiểu sự ánh xạ của nhiều khoá khác nhau vào cùng một địa chỉ (ánh xạ nhiều-một).

Bảng băm với phương pháp nối kết trực tiếp: mỗi địa chỉ của bảng băm(gọi là một bucket) tương ứng một danh sách liên kết.

Các phần tử bị xung đột được nối kết với nhau trên một danh sách liên kết.

Bảng băm với phương pháp nối kết hợp nhất: bảng băm loại này được cài đặt bằng danh sách kề, mỗi phần tử có hai trường: trường *key* chứa khóa của phần tử và trường *next* chỉ phần tử kế bị xung đột. Các phần tử bị xung đột được nối kết nhau qua trường nối kết next.

Bảng băm với phương pháp dò tuyến tính: ví dụ khi thêm phần tử vào bảng băm loại này nếu băm lần đầu bị xung đột thì lần lượt dò địa chỉ kế… cho đến khi gặp địa chỉ trống đầu tiên thì thêm phần tử vào địa chỉ này.

Bảng băm với phương pháp dò bậc hai: ví dụ khi thêm phần tử vào bảng băm loại này, nếu băm lần đầu bị xung đột thì lần lượt dò đến địa chỉ mới, lần dò i ở phần tử cách khoảng i^2 cho đến khi gặp địa chỉ trống đầu tiên thì thêm phần tử vào địa chỉ này.

Bảng băm với phương pháp băm kép: bảng băm loại này dùng hai hàm băm khác nhau, băm lần đầu với hàm băm thứ nhất nếu bị xung đột thì xét địa chỉ khác bằng hàm băm thứ hai.

Ưu điểm của các bảng băm:

Bảng băm là một cấu trúc dung hòa giữa thời gian truy xuất và dung lượng bộ nhớ:

- Nếu không có sự giới hạn về bộ nhớ thì chúng ta có thể xây dựng bảng băm với mỗi khóa ứng với một địa chỉ với mong muốn thời gian truy xuất tức thời.
- Nếu dung lượng bộ nhớ có giới hạn thì tổ chức một số khóa có cùng địa chỉ, lúc này thời gian truy xuất có bi suy giảm đôi chút.

Bảng băm dược ứng dụng nhiều trong thực tế, rất thích hợp khi tổ chức dữ liệu có kích thước lớn và được lưu trữ ở bộ nhớ ngoài.

3.3. VÍ DỤ VỀ CÁC HÀM BĂM

Hàm băm dạng bảng tra:

Hàm băm có thể tổ chức ở dạng bảng tra (còn gọi là bảng truy xuất), thông dụng nhất là ở dạng công thức.

Ví dụ sau đây là bảng tra với khóa là bộ chữ cái, bảng băm có 26 địa chỉ từ 0 đến 25. Khóa a ứng với địa chỉ 0, khoá b ứng với địa chỉ 1,… , z ứng với địa chỉ 25.

Khoá	Địa chỉ	Khóa	Địa chỉ	Khóa	Địa chỉ	Khóa	Địa chỉ
a	0	h	7	o	14	v	21
b	1	I	8	p	15	w	22
c	2	j	9	q	16	x	23
d	3	k	10	r	17	y	24
e	4	l	11	s	18	z	25
f	5	m	12	t	19	/	/
g	6	n	13	u	20	/	/

Hình 3.6 Hàm băm dạng bảng tra được tổ chức dưới dạng danh sách kề.

Hàm băm dạng công thức:

Thông thường hàm băm dạng công thức được xây dựng theo dạng tổng quát f(key).

Người ta thường dùng hàm băm chia dư (modulo) như các ví dụ 1 và 2 sau:

Ví dụ 1: f(key) = key % 10:

Theo ví dụ này, hàm băm f(key) sẽ băm các số nguyên thành 10 địa chỉ khác nhau (ánh xạ vào các địa chỉ từ 0, 1,…, 9). Các khóa có hàng đơn vị là 0 được băm vào địa chỉ 0, các khóa có hàng đơn vị là i (i=0 | 1 | … | 9) được băm vào địa chỉ thứ i.

Ví dụ 2: f(key)=key % M:

Hàm băm loại này cho phép băm các số nguyên thành M địa chỉ khác nhau (ánh xạ vào các địa chỉ từ 0, 1,… M-1).

Ví dụ 3:

Giả sử cần xây dựng một hàm băm với tập khóa số là chuỗi 10 kí tự, tập địa chỉ có M địa chỉ khác nhau .

Có nhiều cách để xây dựng hàm băm này, ví dụ cộng dồn mã ASCII của từng kí tự, sau đó chia dư (modulo) cho M.

Thông thường, hàm băm dạng công thức rất đa dạng và không bị ràng buộc bởi một tiêu chuẩn nào cả.

Yêu cầu đối với hàm băm tốt:

Một hàm băm tốt thường phải thỏa các yêu cầu sau:

· Phải giảm thiểu sự xung đột.

· Phải phân bố đều các phần tử trên M địa chỉ khác nhau của bảng băm.

3.4. CÁC CÁCH GIẢI QUYẾT XUNG ĐỘT

Như đã đề cập ở phần trên, sự xung đột là hiện tượng các khóa khác nhau nhưng băm cùng địa chỉ như nhau, hay ánh xạ vào cùng một địa chỉ.

Một cách tổng quát, khi key1<>key2 mà f(key1)=f(key2) chúng ta nói phần tử có khóa key1 xung đột với phần tử có khóa key2.

Thực tế người ta giải quyết sự xung đột theo hai phương pháp: *phương pháp nối kết* và *phương pháp băm lại*.

Giải quyết sự xung đột bằng phương pháp nối kết:

Các phần tử bị băm cùng địa chỉ (các phần tử bị xung đột) được gom thành một danh sách liên kết. Lúc này mỗi phần tử trên bảng băm cần khai báo thêm trường liên kết next chỉ phần tử kế bị xung đột cùng địa chỉ.

Bảng băm giải quyết sự xung đột bằng phương pháp này cho phép tổ chức các phần tử trên bảng băm rất linh hoạt: khi thêm một phần tử vào bảng băm chúng ta sẽ thêm phần tử này vào danh sách liên kết thích hợp phụ thuộc vào băm. Tuy nhiên bảng bảng băm loại này bị hạn chế về tốc độ truy xuất.

Các loại bảng băm giải quyết sự xung đột bằng phương pháp nối kết như: bảng băm với phương pháp nối kết trực tiếp, bảng băm với phương pháp nối kết hợp nhất.

Giải quyết sự xung đột bằng phương pháp băm lại:

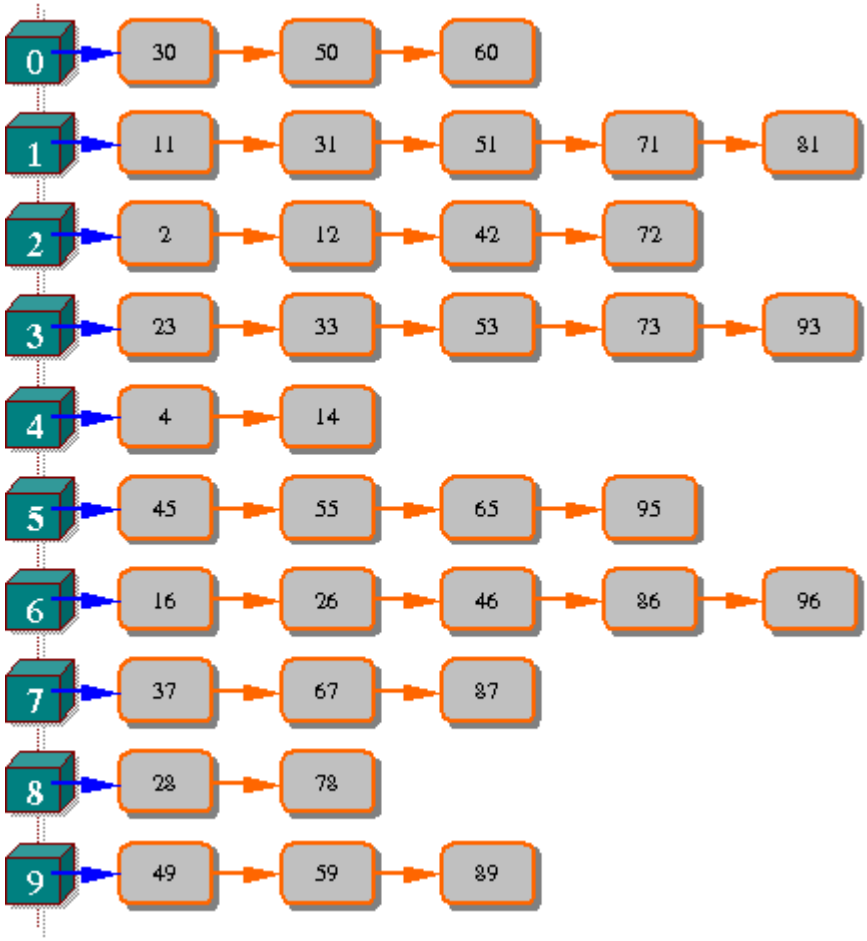
Nếu băm lần đầu bị xung đột thì băm lại lần 1, nếu bị xung đột nữa thì băm lại lần 2,… Quá trình băm lại diễn ra cho đến khi không còn xung đột nữa. Các phép băm lại (rehash function) thường sẽ chọn địa chỉ khác cho các phần tử.

Để tăng tốc độ truy xuất, các bảng băm giải quyết sự xung đột bằng phương pháp băm lại thường được cài đặt bằng danh sách kề. Tuy nhiên việc tổ chức các phần tử trên bảng băm không linh hoạt vì các phần tử chỉ được lưu trữ trên một danh sách kề có kích thước đã xác định trước.

Các loại bảng băm giải quyết sự xung đột bằng phương pháp băm lại như: bảng băm với phương pháp dò tuyến tính, bảng băm với phương pháp dò bậc hai, bảng băm với phương pháp băm kép.

3.4.1. Bảng băm với phương pháp nối kết trực tiếp (Direct chaining Method)

Mô tả: Xem hình vẽ



Hình 3.7. Bảng băm với phương pháp nối kết trực tiếp

Bảng băm được cài đặt bằng các danh sách liên kết, các phần tử trên bảng băm được "băm" thành M danh sách liên kết (từ danh sách 0 đến danh sách M–1). Các phần tử bị xung đột tại địa chỉ i được nối kết trực tiếp với nhau qua danh sách liên kết i. Chẳng hạn, với M=10, các phần tử có hàng đơn vị là 9 sẽ được băm vào danh sách liên kết i = 9.

Khi thêm một phần tử có khóa k vào bảng băm, hàm băm f(k) sẽ xác định địa chỉ i trong khoảng từ 0 đến M-1 ứng với danh sách liên kết i mà phần tử này sẽ được thêm vào.

Khi tìm một phần tử có khóa k vào bảng băm, hàm băm f(k) cũng sẽ xác định địa chỉ i trong khoảng từ 0 đến M-1 ứng với danh sách liên kết i có thể chứa phần tử này. Như vậy, việc tìm kiếm phần tử trên bảng băm sẽ được qui về bài toán tìm kiếm một phần tử trên danh sách liên kết.

Để minh họa cho vấn đề vừa nêu:

Xét bảng băm có cấu trúc như sau:

- Tập khóa K: tập số tự nhiên
- Tập địa chỉ M: gồm 10 địa chỉ (M={0, 1, …, 9})
- Hàm băm f(key) = key % 10.

Hình trên minh họa bảng băm vừa mô tả. Theo hình vẽ, bảng băm đã "*băm*" phần tử trong tập khoá K theo 10 danh sách liên kết khác nhau, mỗi danh sách liên kết gọi là một bucket:

· Bucket 0 gồm những phần tử có khóa tận cùng bằng 0.

· Bucket i(i=0 | … | 9) gồm những phần tử có khóa tận cùng bằng i. Để giúp việc truy xuất bảng băm dễ dàng, các phần tử trên các bucket cần thiết được tổ chức theo một thứ tự,

chẳng hạn từ nhỏ đến lớn theo khóa.

· Khi khởi động bảng băm, con trỏ đầu của các bucket là NULL.

Theo cấu trúc này, với tác vụ insert, hàm băm sẽ được dùng để tính địa chỉ của khoá k của phần tử cần chèn, tức là xác định được bucket chứa phần tử và đặt phần tử cần chèn vào bucket này.

Với tác vụ search, hàm băm sẽ được dùng để tính địa chỉ và tìm phần tử trên bucket tương ứng.

Cài đặt bảng băm dùng phương pháp nối kết trực tiếp :

a.Khai báo cấu trúc bảng băm:

```
#define M   100
struct nodes
{
    int key;
    struct nodes *next;
};
//khai bao kieu con tro chi nut
typedef struct nodes *NODEPTR;
/*
khai bao mang bucket chua M con tro dau cua Mbucket
*/
NODEPTR bucket[M];
```

b.Các phép toán:

Hàm băm

Giả sử chúng ta chọn hàm băm dạng %: f(key)=key % M.

```
int hashfunc (int key)
{
 return (key % M);
}
```

Chúng ta có thể dùng một hàm băm bất kì thay cho hàm băm dạng % trên.

Phép toán initbuckets:

Khởi động các bucket.

```
void initbuckets( )
{
    int b;
    for (b=0;b<M;b++);
    bucket[b]=NULL;
}
```

Phép toán emmptybucket:

Kiểm tra bucket b có bị rỗng không?

```
int emptybucket (int b)
{
    return(bucket[b] ==NULL ?TRUE :FALSE);
}
```

Phép toán emmpty:

Kiểm tra bảng băm có rỗng không?

```
int empty( )
{
    int b;
    for (b = 0;b<M;b++)
    if(bucket[b] !=NULL)  return(FALSE);
    return(TRUE);
```

```
}
```

Phép toán insert:

Thêm phần tử có khóa k vào bảng băm.

Giả sử các phần tử trên các bucket là có thứ tự để thêm một phần tử khóa k vào bảng băm trước tiên chúng ta xác định bucket phù hợp, sau đó đặt phần tử vào vị trí phù hợp trên bucket.

```
void insert(int k)
{
    int i;
    i= hashfunc(k);
    NODEPTR p = new(nodes);
    p->data = k;
    p->next = bucket[i];
    bucket[i] = p;
}
```

Phép toán remove:

Xóa phần tử có khóa k trong bảng băm.

Giả sử các phần tử trên các bucket là có thứ tự, để xóa một phần tử khóa k trong bảng băm cần thực hiện:

- Xác định bucket phù hợp
- Tìm phần tử để xóa trong bucket đã được xác định, nếu tìm thấy phần tử cần xóa thì loại bỏ phần tử theo các phép toán tương tự loại bỏ một phần tử trong danh sách liên kết.

```
void remove ( int k)
{
    int b;
    NODEPTR q, p;
    b = hashfunc(k);
    p = hashbucket(k);
    q=p;
    while(p !=NULL && p->key !=k)
    {
     q=p;
     p=p->next;
    }
    if (p == NULL)
     printf("\n khong co nut co khoa %d" ,k);
    else
    if (p == bucket [b])   pop(b);
    //Tac vu pop cua danh sach lien ket
    else
     delafter(q);
    /*tac vu delafter cua danh sach lien ket*/
}
```

Phép toán clearbucket:

Xóa tất cả các phần tử trong bucket b.

```
void clearbucket (int b)
{
    NODEPTR p,q;
    //q la nut truoc,p la nut sau
    q = NULL;
    p = bucket[b];
```

```
    while(p !=NULL)
    {
      q = p;
      p=p->next;
      freenode(q);
    }
    bucket[b] = NULL; //khoi dong lai butket b
}
```

Phép toán clear:

Xóa tất cả các phần tử trong bảng băm.

```
void clear( )
{
    int b;
    for (b = 0; b<M ; b++)
    clearbucket(b);
}
```

Phép toán traversebucket:

Duyệt các phần tử trong bucket b.

```
void traversebucket (int b)
{
    NODEPTR p;
    p= bucket[b];
    while (p !=NULL)
    {
      printf("%3d", p->key);
      p= p->next;
    }
}
```

Phép toán traverse:

Duyệt toàn bộ bảng băm.

```
void traverse( )
{
    int b;
    for (b = 0;n<M; b++)
    {
      printf("\nButket %d:",b);
      traversebucket(b);
    }
}
```

Phép toán search:

Tìm kiếm một phần tử trong bảng băm,nếu không tìm thấy hàm này trả về hàm NULL,nếu tìm thấy hàm này trả về con trả chỉ tìm phần tử tìm thấy.

```
NODEPTR search(int k)
{
    NODEPTR p;
    int b;
    b = hashfunc (k);
    p = bucket[b];
    while(k > p->key && p !=NULL) p=p->next;
    if (p == NULL | | k !=p->key)// khong tim thay
       return(NULL);
```

```
    else//tim thay
    // else //tim thay
       return(p);
}
```

Nhận xét bảng băm dùng phương pháp nối kết trực tiếp :

Bảng băm dùng phương pháp nối kết trực tiếp sẽ "băm" n phần tử vào danh sách liên kết (M bucket).

Để tốc độ thực hiện các phép toán trên bảng hiệu quả thì cần chọn hàm băm sao cho băm đều n phần tử của bảng băm cho M bucket, lúc này trung bình mỗi bucket sẽ có n/M phần tử. Chẳng hạn, phép toán *search* sẽ thực hiện việc tìm kiếm tuyến tính trên bucket nên thời gian tìm kiếm lúc này có bậc O(n/M) – nghĩa là, nhanh gấp n lần so với việc tìm kiếm trên một danh sách liên kết có n phần tử.

Nếu chọn M càng lớn thì tốc độ thực hiện các phép toán trên bảng băm càng nhanh, tuy nhiên lại càng dùng nhiều bộ nhớ. Do vậy, cần điều chỉnh M để dung hòa giữa tốc độ truy xuất và dung lượng bộ nhớ.

· Nếu chọn M=n thì năng xuất tương đương với truy xất trên mảng (có bậc O(1)), tuy nhiên tốn nhiều bộ nhớ.

· Nếu chọn M =n/k (với k =2,3,4,…) thì ít tốn bộ nhớ hơn k lần (mảng cấp phát ít hơn k lần), nhưng tốc độ chậm đi k lần (xung đột tăng trung bình k lần dẫn đến dãy nối kết nhiều nút hơn)

Chương trình minh họa:

```
#include "stdio.h"
#include "conio.h"
#include "ctype.h"
#include "string.h"
#define M 26

typedef struct tudien
{
 char tu[10];
 char nghia[100];
}tudien;

typedef struct node
{
 tudien data;
 node *link;
}node;

node butket[M];

void khoitao(node b[])
{
 for(int i=0;i<M;i++)
 {
      b[i].link=NULL;
 }
}

int hambam(char tu[])
{
```

```
 char ch;
 ch=toupper(tu[0]);  //ham doi thanh chu In hoa
 return (ch-65)%M;   //65 la ma ASCII cua chu A
}

void themtu(node b[], tudien x)
{
 int i;
 node *p;
 i = hambam(x.tu);
 p = new(node);
 p->data = x;
 p->link = b[i].link;
 b[i].link = p;
}

node *tratu(node b[], char tu[])
{
 int i;
 int tim=0;
 node *p;
 i = hambam(tu);
 p = b[i].link;
 while(p!=NULL && !tim)
 {
      if(strcmp(p->data.tu,tu)==0)
           tim = 1;
      else
           p = p->link;
 }
 if(tim==1)
      return p;
 else
      return NULL;
}

int kiemtra(node b[], char tu[])
{
 int i;
 int tim=0;
 node *p;
 i = hambam(tu);
 p = b[i].link;
 while(p!=NULL && !tim)
 {
      if(strcmp(p->data.tu,tu)==0)
           tim = 1;
      else
           p = p->link;
 }
 if(tim==1)
```

```
            return 1;
      else
            return 0;
    }

    void intudien(node b[])
    {
      int i;
      node *p;
      for(i=0;i<M;i++)
      {
            p=b[i].link;
            printf("\nDanh muc tu %c : ",i+65);
            while(p!=NULL)
            {
                  printf("\nTu : %s, Nghia : %s",p->data.tu, p-
>data.nghia);
                  p = p->link;
            }
      }
    }

    void xoatu(node b[],char tu[])
    {
      int i;
      node *p, *q;
      i=hambam(tu);
      p=b[i].link;
      while(p!=NULL && strcmp(p->data.tu,tu)!=0)
      {
            q = p;
            p=p->link;
      }
      if(p==NULL)
            printf("\nTu nay khong co trong tu dien!!");
      else
      {
            if(p==b[i].link)
            {
                  b[i].link = p->link;
                  delete(p);
            }
            else
            {
                  q->link = p->link;
                  delete(p);
            }
            printf("\nXoa thanh cong!!");
      }
    }
```

```
void ghifile(node b[], char *filename)
{
 FILE *f;
 int i;
 node *p;
 f=fopen(filename,"wb");
 for(i=0;i<M;i++)
 {
      p = b[i].link;
      while(p!=NULL)
      {
           fwrite(&p->data,sizeof(tudien),1,f);
           p=p->link;
      }
 }
 printf("Write File Suscessfull");
 fclose(f);
}

void docfile(node b[], char *filename)
{
 FILE *f;
 tudien tam;
 f=fopen(filename,"rb");
 while(!feof(f))
 {
      fread(&tam,sizeof(tudien),1,f);
      if(kiemtra(b,tam.tu)==0)
           themtu(b,tam);
 }
 printf("Open File Suscessfull");
 fclose(f);
}

void main()
{
 int n, chon;
 tudien x;
 char ch[10];
 node *p;
 do
 {
      printf("\t\nChuong trinh tu dien\n");
      printf("\t1.Nhap tu moi\n");
      printf("\t2.Tra tu\n");
      printf("\t3.In tu dien\n");
      printf("\t4.Xoa 1 tu\n");
      printf("\t5.Ghi File\n");
      printf("\t6.Doc File\n");
      printf("\t7.Thoat");
      printf("\nNhap lua chon cua ban : ");
```

```
          scanf("%d", &chon);
          switch(chon)
          {
          case 1:
               {
                    printf("\nBan nhap bao nhieu tu : ");
                    scanf("%d", &n);
                    for(int i=1;i<=n;i++)
                    {
                         printf("Nhap tu moi : ");
                         fflush(stdin);
                         gets(x.tu);
                         if(kiemtra(butket, x.tu)==1)
                              printf("\nTu  nay  da  co  trong  tu
dien!!\n");
                         else
                         {
                              printf("Nhap nghia cua tu : ");
                              fflush(stdin);
                              gets(x.nghia);
                              themtu(butket,x);
                         }
                    }
                    printf("\nNhan phim bat ki de tiep tuc!");
                    getch();
                    break;
               }
          case 2:
               {
                    printf("\nNhap tu can tra : ");
                    fflush(stdin);
                    gets(ch);
                    p=tratu(butket,ch);
                    if(p==NULL)
                         printf("Tu  nay  khong  co  trong  tu
dien!");
                    else
                         printf("\nTu : %s co nghia la : %s",p-
>data.tu, p->data.nghia);
                    printf("\nNhap phim bat ki de tiep tuc!!");
                    getch();
                    break;
               }
          case 3:
               {
                    intudien(butket);
                    printf("\nNhap phim bat ki de tiep tuc!!");
                    getch();
                    break;
               }
          case 4:
```

```
                {
                        printf("\nNhap tu can xoa : ");
                        fflush(stdin);
                        gets(ch);
                        xoatu(butket,ch);
                        printf("\nNhan phim bat ki de tiep tuc!!");
                        getch();
                        break;
                }
        case 5:
                {
                        ghifile(butket,"D:\\Bangbam.txt");
                        break;
                }
        case 6:
                {
                        docfile(butket,"D:\\Bangbam.txt");
                        break;
                }
        case 7:
                {
                        printf("\nNhap phim bat ki de thoat !!\n");
                        getch();
                        break;
                }
        }
  }while(chon!=7);
}
```

3.4.2. Bảng băm với phương pháp nối kết hợp nhất (Coalesced chaining Method)

Mô tả:

- *Cấu trúc dữ liệu:* Tương tự như trong trường hợp cài đặt bằng phương pháp nối kết trực tiếp, bảng băm trong trường hợp này được cài đặt bằng danh sách liên kết dùng mảng, có M phần tử. Các phần tử bị xung đột tại một địa chỉ được nối kết nhau qua một danh sách liên kết. Mỗi phần tử của bảng băm gồm hai trường:

· Trường key: chứa khóa của mỗi phần tử

· Trường next: con trỏ chỉ đến phần tử kế tiếp nếu có xung đột.

- *Khởi động*: Khi khởi động, tất cả trường key của các phần tử trong bảng băm được gán bởi giá trị Null, còn tất cả các trường next được gán –1.

- *Thêm mới một phần tử*: Khi thêm mới một phần tử có khóa key vào bảng băm, hàm băm f(key) sẽ xác định địa chỉ i trong khoảng từ 0 đến M-1.

· Nếu chưa bị xung đột thì thêm phần tử mới vào địa chỉ này.

· Nếu bị xung đột thì phần tử mới được cấp phát là phần tử trống phía cuối mảng. Cập nhật liên kết next sao cho các phần tử bị xung đột hình thành một danh sách liên kết.

- *Tìm kiếm*: Khi tìm kiếm một phần tử có khóa key trong bảng băm, hàm băm f(key) sẽ giúp giới hạn phạm vi tìm kiếm bằng cách xác định địa chỉ i trong khoảng từ 0 đến M-1, và việc tìm kiếm phần tử khóa có khoá key trong danh sách liên kết sẽ xuất phát từ địa chỉ i.

Để minh họa cho bảng băm với phương pháp nối kết hợp nhất, xét ví dụ sau:

Giả sử, khảo sát bảng băm có cấu trúc như sau:

- Tập khóa K: tập số tự nhiên
- Tập địa chỉ M: gồm 10 địa chỉ (M={0, 1, …, 9}

- Hàm băm f(key) = key % 10.

key	next
NULL	-1
NULL	-1
...	...
NULL	-1

0	NULL	-1
1	A	M-1
2	C	-1
3	E	-1
...	...	
M-2	D	-1
M-1	B	M-2

Hình 3.8. Bảng khởi tạo và kết quả bảng băm với phương pháp nối kết hợp nhất với Key : A C B D E, vị trí băm tương ứng: 1 2 1 1 3

Cài đặt bảng băm dùng phương pháp nối kết hợp nhất:

a. Khai báo cấu trúc bảng băm:

```
#define NULLKEY -1
#define M 100
/*
M la so nut co tren bang bam, du de chua cac nut nhap vao bang bam
*/
//Khai bao cau truc mot nut cua bang bam
struct node
{
    int key; //khoa cua nut tren bang bam
    int next; //con tro chi nut ke tiep khi co xung dot
};
//Khai bao bang bam
struct node hashtable[M];
int avail;
/*
bien toan cuc chi nut trong o cuoi table duoc cap nhat khi co xung dot
*/
```

b. Các phép toán:

Hàm băm:

Giả sử chúng ta chọn hàm băm dạng modulo: f(key)=key % 10.

```
int hashfunc(int key)
{
 return(key % 10);
}
```

Chúng ta có thể dùng một hàm băm bất kì thay cho hàm băm dạng % trên.

Phép toán khởi tạo (Initialize):

Phép toán này cho khởi động bảng băm: gán tất cả các phần tử trên bảng có trường key là NULL, trường next là –1.

Gán biến toàn cục avail=M-1, là phần tử cuối danh sách chuẩn bị cấp phát nếu xãy ra xung đột.

```
void initialize()
{
int i;
for(i = 0;i<M;i++)
{
```

```
        hashtable[i].key = NULLKEY;
        hashtable[i].key = -1;
    }
    avail =M-1;
    /* nut M-1 la nut o cuoi bang chuan bi cap phat neu co xung
dot*/
    }
```

Phép toán kiểm tra rỗng (empty):

Kiểm tra bảng băm có rỗng không.

```
int empty ();
{
    int i;
    for(i = 0;i< M;i++)
    if(hashtable[i].key !=NULLKEY)
      return(FALSE);
    return(TRUE);
}
```

Phép toán tìm kiếm (search):

Tìm kiếm theo phương pháp tuyến tính, nếu không tìm thấy hàm tìm kiếm trả về trị M, nếu tìm thấy hàm này trả về địa chỉ tìm thấy.

```
int search(int k)
{
    int i;
    i=hashfunc(k);
    while(k !=hashtable[i].key && i !=-1)
      i=hashtable[i].next;
    if(k== hashtable[i]key)
      return(i);//tim thay
    return(M);//khong tim thay
}
```

Phép toán lấy phần tử trống (Getempty):

Chọn phần tử còn trống phía cuối bảng băm để cấp phát khi xảy ra xung đột.

```
int getempty()
{
    while(hashtable[avail].key !=NULLKEY)
      avail - -;
    return(avail);
}
```

Phép toán chèn phần tử mới vào bảng băm (insert):

Thêm phần tử có khóa k vào bảng băm.

```
int insert(int k)
{
    int i; //con tro lan theo danh sach lien ket chua cac nut bi xung dot
    int j; //dia chi nut trong duoc cap phat
    i = search(k);
    if(i !=M)
    {
      printf("\n khoa %d bi tRung,khong them nut nay duoc",k);
      return(i);
    }
    i=hashfunc(k);
```

```
        while(hashtable[i]next >=0) i=hashtable[i].next;
        if(hashtable[i].key == NULLKEY)
          //Nut i con trong thi cap nhat
          j = i;
        else
        //Neu nut i la nut cuoi cua DSLK
        {
          j = getempty();
          if(j < 0)
          {
                printf("\n Bang bam bi day,khongthem nut co khoa %d duoc"k);
                return(j);
          }
          else
              hashtable[i].next = j;
        }
        hashtable[j].key = k;
        return(j);
    }
```

Nhận xét bảng băm dùng phương pháp nối kết hợp nhất:

Thực chất cấu trúc bảng băm này chỉ tối ưu khi băm đều, nghĩa là mỗi danh sách liên kết chứa một vài phần tử bị xung đột, tốc độ truy xuất lúc này có bậc O(1). Trường hợp xấu nhất là băm không đều vì hình thành một danh sách có n phần tử nên tốc độ truy xuất lúc này có bậc O(n).

Chương trình minh họa:

Chương trình Hashtable, dùng phương pháp nối kết hợp nhất *(coalesced chaining method*) - Cài đặt bằng danh sách kề.

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <conio.h>
    #define TRUE 1
    #define FALSE 0
    #define NULLKEY -1
    #define M 100
    /* M la so nut co tren bang bam,du de chua cac nut nhap vao
bang bam */
    //Khai bao cau truc mot nut cua bang bam
    struct node
    {
        int key; //khoa cua nut tren bang bam
        int next; //con tro chi nut ke tiep khi co xung dot
    };
    //Khai bao bang bam
    struct node hashtable[M];
    int avail;
    //bien toan cuc chi nut trong o cuoi table duoc cap phat
khi co xung dot
    //Ham bam
    int hashtable(int key)
    {
        return(key % M);
```

```
}
//Khoi dong bang bam
void initialize()
{
    int i;
    for (i=0; i<M;i++)
    {
     hashtable[i].key=NULLKEY;
     hashtable[i].next=-1;
    }
    avail=M-1; //nut M-1 la nut o cuoi bang chuan bi cap
phat nut co xung dot
}

//Tac vu empty: kiem tra bang baam co ranh khong
int empty()
{
    int i;
    for (i= 0;i<M;i++)
     if(hashtable[i].key !=NULLKEY) return(FALSE);
    return(TRUE);
}

/*Tac vu search: tim kiem theom phuong phap tuyen tinh ,
neu khong tim thay ham nay tra ve vi tri M, neu tim thay ham
nay tra ve dia chi tim thay */
int search(int k)
{
    int i;
    i= hashfunc(k);
    while(k !=hashtable[i].key && 1 !=-1)
     i = hashtable[i].next;
    if(k == hashtable[i].key;
     //Tim thay
     return(i);
    else
     //khong tim thay
     return(M);
}

/*Ham getempty: chon nut con trong phia cuoi hashtable de
cap nhat khi xay ra xung dot
*/
int getempty()
{
    while(hashtable[avail].key !=NULLKEY)avail--;
    return(avail);
}

//Tac vu insert: them nut co khoa k vao bang bam
int insert (int k)
```

```
{
    int i;
    //con tro lan theo danh sach lien ket chua cac nut //bi
xung dot
    int j;
    //dia chi nut trong duoc cap phat
    i = search(k);
    if(i !=M)
    {
     printf("\n khoa %d bi tRung, khong them nut nay duoc",
 k);
     return(i);
    }
    i = hashfunc(k0;
    while(hashtablr[i].next >=0) i = hashtable[i].next;
    if(hashtable[i].key ==NULLKEY)
     //Neu nut i con trong thi cap phat
     j=i;
    else
    {
     //Neu nut i la nut cuoi cua DSLK
     j=getempty();
     if(j < 0)
     {
      printf("\n Bang bam bi day khong them nut co khoa % d
   duoc:", k);
      return(j);
     }
     else hashtable[i],next=j;
    }
    hashtable[j].key=k;
    return(j);
}
//Tac vu viewtable:xem chi tiet bang bam
void viewtable()
{
    int i;
    for(i= 0;i < M; i++)
     printf("\ntable[%2d]: %4d",i,hashtable[i].next);
}
//Chuong trinh chinh
void main( )
{
    int i,n,p,q;
    int b,key,chucnang;
    char c;
    clrscr();
    //Khoi dong bang bam
    initialize();
    do
    {
```

```
//Menu chinh cua chuong trinh
printf("\n\nCac chuc nang cua chuong trinh:\n");
printf("1: Them nut moi vao bang bam\n");
printf("2: Them ngau nhien nut vao bang bam\n");
printf("3: Xoa toan bo bang bam\n");
printf("4: Xem chi tiet bang bam\n");
printf("5 : Tim kiem tren bang bam\n");
printf("0: Ket thuc chuong trinh\n");
printf("\nChuc nang ban chon:");
scanf("%d", & chucnang);
switch(chucnang)
{
case 1:{
    printf("\nTHEM NUT MOI VAO BANG BAM");
    printf("\n Khoa cua nut moi:");
    scanf("%d",&key);
    insert(key);
    break;
}
case 2:{
    printf("\n Them ngau nhien nut vao bang bam");
    printf(\n Ban muon them bao nhieu nut:");
    scanf("%d",&key);
    for(i=0;i<n;i++)
    {
      key=random(1000);
      insert(key);
    }
    beark;
}
case 3: {
    printf("\n XOA TOAN BO BANG BAM");
    printf("\N BAN CO CHAC CHAN KHONG (C/K):");
    c=getch();
    if(c==’c’ | | c ==’c’)
      initialize( );
    beark;
}
case 4:{
    printf("\n XEM CHI TIET BANMG BAM:");
    viewtable();
    break;
}
case 5:{
    printf("\nTIM KIEM TREN BANG BAM:");
    printf("\n Khoa can tim:");
    scanf("%d",&key);
    if(search(key0=M)
      printf("khongtim thay");
    else
      printf("Tim thay tai dia chi %d trong bang bam",
```

```
            search(ke y )) ;
            beark;
        }
        }
    }while(chucnang !=0);
}
```

3.4.3. Bảng băm với phương pháp dò tuyến tính (Linear Probing Method)

Mô tả:

- *Cấu trúc dữ liệu:* Bảng băm trong trường hợp này được cài đặt bằng danh sách kề có M phần tử, mỗi phần tử của bảng băm là một mẫu tin có một trường key để chứa khoá của phần tử.

Khi khởi động bảng băm thì tất cả trường key được gán NULL

- *Khi thêm phần tử* có khoá key vào bảng băm, hàm băm f(key) sẽ xác định địa chỉ i trong khoảng từ 0 đến M-1:

· Nếu chưa bị xung đột thì thêm phần tử mới vào địa chỉ này.

· Nếu bị xung đột thì hàm băm lại lần 1, hàm f_1 sẽ xét địa chỉ kế tiếp, nếu lại bị xung đột thì hàm băm lại lần 2, hàm f_2 sẽ xét địa chỉ kế tiếp nữa, …, và quá trình cứ thế cho đến khi nào tìm được địa chỉ trống và thêm phần tử mới vào địa chỉ này.

- *Khi tìm một phần tử* có khoá key trong bảng băm, hàm băm f(key) sẽ xác định địa chỉ i trong khoảng từ 0 đến M-1, tìm phần tử khoá key trong khối đặt chứa các phần tử xuất phát từ địa chỉ i.

Hàm băm lại của phương pháp dò tuyến tính là truy xuất địa chỉ kế tiếp. Hàm băm lại lần i được biểu diễn bằng công thức sau:

f(key)=(f(key)+i) %M với f(key) là hàm băm chính của bảng băm.

Lưu ý địa chỉ dò tìm kế tiếp là địa chỉ 0 nếu đã dò đến cuối bảng.

Giả sử, khảo sát bảng băm có cấu trúc như sau:

- Tập khóa K: tập số tự nhiên
- Tập địa chỉ M: gồm 10 địa chỉ (M={0, 1, …, 9}
- Hàm băm f(key) = key % 10.

0	NULL	0	NULL	0	NULL	0	NULL	0	56
1	NULL	1	NULL	1	NULL	1	NULL	1	NULL
2	32	2	32	2	32	2	32	2	32
3	53	3	53	3	53	3	53	3	53
4	NULL	4	22	4	22	4	22	4	22
5	NULL	5	92	5	92	5	92	5	92
6	NULL	6	NULL	6	34	6	34	6	34
7	NULL	7	NULL	7	17	7	17	7	17
8	NULL	8	NULL	8	NULL	8	24	8	24
9	NULL	9	NULL	9	NULL	9	37	9	37

Hình 3.9 Bảng thể hiện thêm các nút 32, 53, 22, 92, 17, 34, 24, 37, 56 vào bảng băm.

Cài đặt bảng băm dùng phương pháp dò tuyến tính:

a. Khai báo cấu trúc bảng băm:

```
#define NULLKEY -1
#define M 100
/* M la so nut co tren bang bam,du de chua cac nut nhap vao
bang bam */
//khai bao cau truc mot nut cua bang bam
struct node
{
    int key; //khoa cua nut tren bang bam
};
//Khai bao bang bam co M nut
struct node hashtable[M];
int NODEPTR;
/*bien toan cuc chi so nut hien co tren bang bam*/
```

b. Các tác vụ:

Hàm băm:

Giả sử chúng ta chọn hàm băm dạng%:f(key0=key %10.

```
int hashfunc(int key)
{
 return(key% 10);
}
```

Chúng ta có thể dùng một hàm băm bất kì thay cho hàm băm dạng % trên.

Phép toán khởi tạo (initialize):

Khởi tạo bảng băm.

Gán tất cả các phần tử trên bảng có trường key là NULL.

Gán biến toàn cục N=0.

```
void initialize( )
{
    int i;
    for(i=0;i<M;i++)
     hashtable[i].key=NULLKEY;
    N=0;
    //so nut hien co khoi dong bang 0
}
```

Phép toán kiểm tra trống (empty):

Kiểm tra bảng băm có trống hay không.

```
int empty( );
{
    return(N==0 ? TRUE;FALSE);
}
```

Phép toán kiểm tra đầy (full):

Kiểm tra bảng băm đã đầy chưa.

```
int full( )
{
    return (N==M-1 ? TRUE; FALSE);
}
```

Lưu ý bảng băm đầy khi N=M-1, chúng ta nên dành ít nhất một phần tử trống trên bảng băm.

Phép toán search:

Việc tìm kiếm phần tử có khoá k trên một khối đặc, bắt đầu từ một địa chỉ i = HF(k), nếu không tìm thấy phần tử có khoá k, hàm này sẽ trả về trị M, còn nếu tìm thấy, hàm này trả về địa chỉ tìm thấy.

```
int search(int k)
{
    int i;
    i=hashfunc(k);
    while(hashtable[i].key!=k && hashtable[i].key !=NULKEY)
    {
    //bam lai (theo phuong phap do tuyen tinh:fi(key) =
f(key)+1) % M)
     i=i+1;
     if(i>=M)
       i=i-M;
    }
    if(hashtable[i].key==k)
     //tim thay
     return(i);
    else
     //khong tim thay
     return(M);
}
```

Phép toán insert:

Thêm phần tử có khoá k vào bảng băm.

```
int insert(int k)
{
    int i, j;
    if(full( ))
    {
     printf("\n Bang bam bi day khong them nut co khoa %d
 duoc",k);
     return;
    }
    i=hashfunc(k);
    while(hashtable[i].key !=NULLKEY)
    {
     //Bam lai (theo phuong phap do tuyen tinh)
     i ++;
     if(i >M)  i= i-M;
    }
    hashtable[i].key=k;
    N=N+1;
    return(i);
}
```

Nhận xét bảng băm dùng phương pháp dò tuyến tính:

Bảng băm này chỉ tối ưu khi băm đều, nghĩa là, trên bảng băm các khối đặc chứa vài phần tử và các khối phần tử chưa sử dụng xen kẻ nhau, tốc độ truy xuất lúc này có bậc 0(1). Trường hợp xấu nhất là băm không đều hoặc bảng băm đầy, lúc này hình thành một khối đặc có n phần tử, nên tốc độ truy xuất lúc này có bậc 0(n).

Chương trình minh họa:

Bảng băm, dùng phương pháp dò tuyến tính (linear proping method)-cài đặt bằng danh

sách kề.

```
#include "stdio.h"
#include "stdlib.h"
#include "conio.h"
#define TRUE 1
#define FALSE 0
#define NULLKEY -1
#define M 10

//Khai bao cau truc mot nut cua bang bam
struct node
{
 int key;  //khoa cua nut tren bam
};

//Khai bao bang bam co M nut
struct node hashtable[M];
int N; //bien toan cuc chi so nut hien co tren bang bam

//ham bam
int hashfunc(int key)
{
 return(key % M);
}

//Khoi dong bang bam
void initialize()
{
 int i;
 for(i=0;i<M;i++)
      hashtable[i].key = NULLKEY;
 N=0; // so nut hien co khoi dong bang 0
}

//Tac vu empty : kiem tra ca bang bam co rong hay khong
int empty()
{
 return (N==0 ? TRUE : FALSE);
}

//Tac vu full : kiem tra bang bam co day chua
int full()
{
 return (N==M-1 ? TRUE : FALSE);
}

//Tac vu search : tim kiem nut co khoa k tren bang bam
int search(int k)
{
 int i;
 i=hashfunc(k);
```

```
     while(hashtable[i].key!=k && hashtable[i].key!=NULLKEY)
     {
          //bam   lai(theo   phuong   phap   do   tuyen   tinh):
hi(key)=h(key)+i % M
          i=i+1;
          if(i>=M)
               i=i-M;
     }
     if(hashtable[i].key==k)  //tim thay
          return(i);
     else
          return(M);     //khong tim thay
    }

    //tac vu insert : them nut co khoa k vao bang bam
    int insert(int k)
    {
     int i,j;
     if(full())
     {
          printf("Bang bam bi day khong the them nut co khoa %d
duoc",k);
          return 0;
     }
     if(search(k)<M)
     {
          printf("So nay da co trong bang bam");
          return 0;
     }
     i=hashfunc(k);
     while(hashtable[i].key!=NULLKEY)
     {
          //bam lai theo phuong phap tuyen tinh
          i++;
          if(i>M)
               i=i-M;
     }
     hashtable[i].key=k;
     N=N+1;
     return(i);
    }

    //tac vu remove : xoa nut tai dia chi i tren bang bam
    void remove(int i)
    {
     int j, r, cont, a;
     cont= TRUE;
     do
     {
          hashtable[i].key=NULLKEY;
          j=i;
```

```
          do
          {
               i=i+1;
               if(i>=M)
                    i=i-M;
               if(hashtable[i].key==NULLKEY)
                    cont=FALSE;
               else
               {
                    r=hashfunc(hashtable[i].key);
                    a=(j<r && r<=i || (r<=i && i<j) || (i<j &&
j<r);
               }
          }while(cont && a);
          if(cont)
               hashtable[j].key=hashtable[i].key;
     }while(cont);

    }

    //tac vu viewtable : xem chi tiet bang bam
    void viewtable()
    {
     int i;
     for(i=0;i<M;i++)
          printf("\nTable[%2d] : %4d",i,hashtable[i].key);
    }

    //chuong trinh chinh
    void main()
    {
     int i,n,p,q;
     int b, key, chucnang;
     char c;
     //khoi tao bang bam
     initialize();
     do
     {
          //menu chinh cua chuong tinh
          printf("\t\n\nCac chuc nang chinh cua chuong trinh :
\n");
          printf("\t1.Them nut moi vao bang bam\n");
          printf("\t2.Them ngau nhien mot nut vao bang bam\n");
          printf("\t3.Xoa nut tren bang bam\n");
          printf("\t4.Xoa toan bo bang bam\n");
          printf("\t5.Xem chi tiet bang bam\n");
          printf("\t6.Tim kiem tren bang bam\n");
          printf("\t0.Ket thuc chuong trinh\n");
          printf("\tChuc nang ban chon : ");
          scanf("%d",&chucnang);
          switch(chucnang)
```

```
          {
          case 1:
               {
                    printf("\nThem nut vao bang bam ");
                    printf("\nKhoa cua nut moi : ");
                    scanf("%d", &key);
                    insert(key);
                    break;
               }
          case 2:
               {
                    randomize();
                    printf("\nThem  ngau  nhien  nhieu  nut  vao
bang bam");
                    printf("\nBan muon them bao nhieu nut : ");
                    scanf("%d", &n);
                    for(i=0;i<n;i++)
                    {
                         key=random(1000);
                         insert(key);
                    }
                    break;
               }
          case 3:
               {
                    printf("\nXoa nut tren bang bam");
                    printf("\nkhoa cua nut can xoa : ");
                    scanf("%d", &key);
                    i=search(key);
                    if(i==M)
                         printf("Khong  co  nut  voi  khoa  can
xoa");
                    else
                    {
                         remove(i);
                         N--;
                    }
                    break;
               }
          case 4:
               {
                    printf("\nXoa toan bo bang bam");
                    printf("\nBan co chat khong (c/k) : ");
                    c=getch();
                    if(c=='c'||c=='C')
                         initialize();
                    break;
               }
          case 5:
               {
                    printf("\nXem chi tiet bang bam");
```

```
                    viewtable();
                    break;
               }
          case 6:
               {
                    printf("Tim kiem tren bang bam");
                    printf("Khoa can tim : ");
                    if(search(key)==M)
                         printf("\nKhong tim thay");
                    else
                    {
                         printf("Tim thay tai dia chi %d trong
bang bam",search(key));
                         break;
                    }
               }
               scanf("%d",&key);
          }
      }while(chucnang!=0);
    }
```

3.4.4. Bảng băm với phương pháp dò bậc hai (Quadratic Probing Method)

Mô tả:

- *Cấu trúc dữ liệu:* Bảng băm dùng phương pháp dò tuyến tính bị hạn chế do rải các phần tử không đều, bảng băm với phương pháp dò bậc hai rải các phần tử đều hơn.

Bảng băm trong trường hợp này được cài đặt bằng danh sách kề có M phần tử, mỗi phần tử của bảng băm là một mẫu tin có một trường key để chứa khóa các phần tử.

- *Khi khởi động* bảng băm thì tất cả trường key bị gán NULL.

Khi thêm phần tử có khóa key vào bảng băm, hàm băm f(key) sẽ xác định địa chỉ i trong khoảng từ 0 đến M-1.

· Nếu chưa bị xung đột thì thêm phần tử mới vào địa chỉ i này.

· Nếu bị xung đột thì hàm băm lại lần 1, f1 sẽ xét địa chỉ hiện tại cộng với 1^2, nếu lại bị xung đột thì hàm băm lại lần 2, f2 sẽ xét địa chỉ hiện tại cộng với 2^2 ,… , quá trình cứ thế cho đến khi nào tìm được trống và thêm phần tử vào địa chỉ này.

- *Khi tìm kiếm* một phần tử có khóa key trong bảng băm thì xét phần tử tại địa chỉ i=f(key), nếu chưa tìm thấy thì xét phần tử có địa chỉ bằng địa chỉ hiện tại cộng với 1^2, 2^2, ..., quá trình cứ thế cho đến khi tìm được khóa (trường hợp tìm thấy) hoặc rơi vào địa chỉ trống (trường hợp không tìm thấy).

- *Hàm băm* lại của phương pháp dò bậc hai là truy xuất các địa chỉ cách bậc 2. Hàm băm lại hàm i được biểu diễn bằng công thức sau:

fi(key)=(f(key) + i^2) % M

với f(key) là hàm băm chính của bảng băm.

Nếu đã dò đến cuối bảng thì trở về dò lại từ đầu bảng.

Bảng băm với phương pháp do bậc hai nên chọn số địa chỉ M là số nguyên tố.

Bảng băm minh họa có cấu trúc như sau:

- Tập khóa K: tập số tự nhiên
- Tập địa chỉ M: gồm 10 địa chỉ (M={0, 1, …, 9}
- Hàm băm f(key) = key % 10.

Cài đặt bảng băm dùng phương pháp dò bậc hai:

a. Khai báo cấu trúc bảng băm:

```
#define NULLKEY -1
#define M 101
```

```
    /* M la so nut co tren bang bam,du de chua cac nut nhap vao
bang bam,chon M la so nguyen to */
    //Khai bao nut cua bang bam
    struct node
    {
        int key; //Khoa cua nut tren bang bam
    };
    //Khai bao bang bam co M nut
    struct node hashtable[M];
    int N;
    //Bien toan cuc chi so nut hien co tren bang bam
```

b. Các tác vụ :

Hàm băm:

Giả sử chúng ta chọn hàm băm dạng%: f(key)=key %10.

```
int hashfunc(int key)
{
    return(key% 10);
}
```

Chúng ta có thể dùng một hàm băm bất kì tahy cho hàm băm dạng % trên.

Phép toán initialize

Khởi động hàm băm.

Gán tất cả các phần tử trên bảng có trường key là NULLKEY.

Gán biến toàn cục N=0.

```
void initialize()
{
    int i;
    for(i=0; i<M;i++)  hashtable[i].key = NULLKEY;
    N=0; //so nut hien co khoi dong bang 0
}
```

Phép toán empty:

Kiểm tra bảng băm có rỗng không

```
int empty()
{
    return(N ==0 ?TRUE :FALSE);
}
```

Phép toán full:

Kiểm tra bảng băm đã đầy chưa .

```
int full()
{
    return(N = = M-1 ?TRUE :FALSE);
}
```

Lưu ý bảng băm đầy khi N=M-1 chúng ta nên chừa ít nhất một phần tử trong trên bảng băm!

Phép toán search:

Tìm phần tử có khóa k trên bảng băm,nếu không tìm thấy hàm này trả về trị M, nếu tìm thấy hàm này trả về địa chỉ tìm thấy.

```
int search(int k)
{
    int i, d;
    i = hashfuns(k);
    d = 1;
```

```
        while(hashtable[i].key!=k&&hashtable[i].key !=NULLKEY)
        {
            //Bam lai (theo phuong phap bac hai)
            i = (i+d*d) % M;
            d ++;
        }
        hashtable[i].key =k;
        N = N+1;
        return(i);
    }
```

Nhận xét bảng băm dùng phương pháp dò bậc hai:

Nên chọn số địa chỉ M là số nguyên tố. Khi khởi động bảng băm thì tất cả M trường key được gán NULL, biến toàn cục N được gán 0.

Bảng băm đầy khi N = M-1, và nên dành ít nhất một phần tử trống trên bảng băm.

Bảng băm này tối ưu hơn bảng băm dùng phương pháp dò tuyến tính do rải rác phần tử đều hơn, nếu bảng băm chưa đầy thì tốc độ truy xuất có bậc 0(1). Trường hợp xấu nhất là bảng băm đầy vì lúc đó tốc độ truy xuất chậm do phải thực hiện nhiều lần so sánh.

3.4.5. Bảng băm với phương pháp băm kép (Double hashing Method)

Mô tả:

- *Cấu trúc dữ liệu:* Bảng băm này dùng hai hàm băm khác nhau với mục đích để rải rác đều các phần tử trên bảng băm.

Chúng ta có thể dùng hai hàm băm bất kì, ví dụ chọn hai hàm băm như sau:

f1(key)= key %M.

f2(key) =(M-2)-key %(M-2).

Bảng băm trong trường hợp này được cài đặt bằng danh sách kề có M phần tử, mỗi phần tử của bảng băm là một mẫu tin có một trường key để lưu khoá các phần tử.

- *Khi khởi động* bảng băm,tất cả trường kay được gán NULL.
- *Khi thêm phần tử* có khoá key vào bảng băm, thì i=f1(key) và j=f2(key) sẽ xác định địa chỉ i và j trong khoảng từ 0 đến M-1:

· Nếu chưa bị xung đột thì thêm phần tử mới tại địa chỉ i này.

· Nếu bị xung đột thì hàm băm lại lần 1, f1 sẽ xét địa chỉ mới i+j, nếu lại bị xung đột thì hàm băm lại lần 2 f2 sẽ xét địa chỉ i+2j, …, quá trình cứ thế cho đến khi nào tìm được địa chỉ trống và thêm phần tử vào địa chi này.

- *Khi tìm kiếm* một phần tử có khoá key trong bảng băm, hàm băm i=f1(key) và j=f2(key) sẽ xác định địa chỉ i và j trong khoảng từ 0 đến M-1. Xét phần tử tại địa chỉ i, nếu chưa tìm thấy thì xét tiếp phần tử i+j, i+2j, …, quá trình cứ thế cho đến khi nào tìm được khoá (trường hợp tìm thấy) hoặc bị rơi vào địa chỉ trống (trường hợp không tìm thấy).

Bảng băm dùng hai hàm băm khác nhau, hàm băm lại của phương pháp băm kép được tính theo i (từ hàm băm thứ nhất) và j (từ hàm băm thứ hai) theo một công thức bất kì, ở đây minh họa bằng địa chỉ mới cách j. Nếu đã dò đến cuối bảng thì trở về dò lại từ đầu bảng.

Bảng băm với phương pháp băm kép nên chọn số địa chỉ M là số nguyên tố.

Bảng băm minh họa có cấu trúc như sau:

- Tập khóa K: tập số tự nhiên
- Tập địa chỉ M: gồm 10 địa chỉ (M={0, 1, …, 9})
- Hàm băm f(key) = key % 10.

Minh hoạ:

Sau đây là minh hoạ cho bảng băm có tập khóa là tập số tự nhiên, tập địa chỉ có 11 địa chỉ (M=11)(từ địa chỉ 0 đến 10),chọn hàm băm f1(key)=key % 10 và f2(key)=9-key %9.

Xem việc minh hoạ này như một bài tập.

Cài đặt bảng băm dùng phương pháp băm kép:

a. Khai báo cấu trúc bảng băm:

```
#define NULLKEY -1
#define M 101
/*M la so nut co tren bang bam,du de chua cac nut nhap vao
bang bam,chon M la so nguyen to
*/
//Khai bao phan tu cua bang bam
struct node
{
    int key;//khoa cua nut tren bang bam
};
//khai bao bang bam co M nut
struct node hashtable[M];
int N;
//bien toan cuc chi so nut hien co tren bang bam
```

b. Các tác vụ:

Hàm băm:

Giả sử chúng ta chọn hai hàm băm dạng %:

f1(key0=key %M va f2(key) =M-2-key%(M-2).

```
//Ham bam thu nhat
int hashfunc1(int key)
{
 return(key%M);
}
//Ham bam thu hai
int hashfunc2(int key)
{
 return(M-2 - key%(M-2));
}
```

Chúng ta có thể dùng hai hàm băm bất kỳ thay cho hai hàm băm dạng % trên.

Phép toán initialize :

Khởi động bảng băm.

Gán tất cả các phần tử trên bảng có trường key là NULL.

Gán biến toàn cục N = 0.

```
void initialize()
{
    int i;
    for (i = 0 ; i<M ; i++ )
      hashtable [i].key = NULLKEY;
    N = 0;// so nut hien co khoi dong bang 0
}
```

Phép toán empty :

Kiểm tra bảng băm có rỗng không.

```
int empty() .
{
    return (N == 0 ? TRUE : FALSE) ;
}
```

Phép toán full :

Kiểm tra bảng băm đã đầy chưa.

```
int full() .
```

```
{
    return (N == M-1 ? TRUE : FALSE) ;
}
```

Lưu ý bảng băm đầy khi N=M-1, chúng ta nên dành ít nhất một phần tử trống trên bảng băm.

Phép toán search :

Tìm kiếm phần tử có khóa k trên bảng băm, nếu không tìm thấy hàm này trả về về trị M, nếu tìm thấy hàm này trả về địa chỉ tìm thấy.

```
int search(int k)
{
    int i, j ;
    i = hashfunc (k);
    j = hashfunc2 (k);
    While (hashtable [i].key!=k && hashtable [i] .key ! =
NULLKEY)
    //bam lai (theo phuong phap bam kep)
     i = (i+j) % M ;
    if (hashtable [i]).key == k)
     // tim thay
     return (i) ;
    else
     // khong tim thay
     return (M) ;
}
```

Phép toán insert :

Thêm phần tử có khoá k vào bảng băm.

```
int insert(int k)
{
    int i, j;
    if (full () )
    {
     printf ("Bang bam bi day") ;
     return (M) ;
    }
    if (search (k) < M)
    {
     printf ("Da co khoa nay trong bang bam") ;
     return (M) ;
    }
    i = hashfunc (k) ;
    j = hashfunc 2 (k) ;
    while (hashtable [i].key ! = NULLEY)
     // Bam lai (theo phuong phap bam kep)
     i = (i + j) % M;
    hashtable [i].key = k ;
    N = N+1;
    return (i) ;
}
```

Nhận xét bảng băm dùng phương pháp băm kép:

Nên chọn số địa chỉ M là số nguyên tố.

Bảng băm đầy khi N = M-1, chúng ta nên dành ít nhất một phần tử trống trên bảng băm.

Bảng băm được cài đặt theo cấu trúc này linh hoạt hơn bảng băm dùng phương pháp dò tuyến tính và bảng băm dùng phương pháp sò bậc hai, do dùng hai hàm băm khác nhau nên việc rải phần tử mang tính ngẫu nhiên hơn, nếu bảng băm chưa đầy tốc độ truy xuất có bậc O(1). Trường hợp xấu nhất là bảng băm gần đầy, tốc độ truy xuất chậm do thực hiện nhiều lần so sánh.

Chương trình minh họa:

```
#include "stdio.h"
#include "stdlib.h"
#include "conio.h"

#define TRUE 1
#define FALSE 0
#define NULLKEY -1
#define M 10

//Khai bao cau truc mot nut cua bang bam
struct node
{
 int key;  //khoa cua nut tren bam
};

//Khai bao bang bam co M nut
struct node hashtable[M];
int N; //bien toan cuc chi so nut hien co tren bang bam

//ham bam
int hashfunc(int key)
{
 return(key % M);
}

//ham bam thu hai
int hashfunc2(int key)
{
 return (M-2-key%(M-2));
}

//Khoi dong bang bam
void initialize()
{
 int i;
 for(i=0;i<M;i++)
      hashtable[i].key = NULLKEY;
 N=0; // so nut hien co khoi dong bang 0
}

//Tac vu empty : kiem tra ca bang bam co rong hay khong
int empty()
{
 return (N==0 ? TRUE : FALSE);
}
```

```
    //Tac vu full : kiem tra bang bam co day chua
    int full()
    {
      return (N==M-1 ? TRUE : FALSE);
    }

    //Tac vu search : tim kiem nut co khoa k tren bang bam
    int search(int k)
    {
      int i,j;
      i=hashfunc(k);
      j=hashfunc2(k);
      while(hashtable[i].key!=k && hashtable[i].key!=NULLKEY)
            //bam lai(theo phuong phap bam kep
            i=(i+j)%M;
      if(hashtable[i].key==k)  //tim thay
            return(i);
      else
            return(M);

    }

    //tac vu insert : them nut co khoa k vao bang bam
    int insert(int k)
    {
      int i,j;
      if(full())
      {
            printf("Bang bam bi day khong the them nut co khoa %d
duoc",k);
            return(M);
      }
      if(search(k)<M)
      {
            printf("So nay da co trong bang bam");
            return(M);
      }
      i=hashfunc(k);
      j=hashfunc2(k);
      while(hashtable[i].key!=NULLKEY)
            //bam lai theo phuong phap tuyen tinh
            i=(i+j)%M;
      hashtable[i].key=k;
      N=N+1;
      return(i);
    }

    //tac vu remove : xoa nut tai dia chi i tren bang bam
    void remove(int i)
    {
```

```
     int j, r, cont, a;
     cont= TRUE;
     do
     {
          int h = hashfunc2(hashtable[i].key);
          j=i;
          do
          {
               i=(i+h)%M;
               if(hashtable[i].key==NULLKEY)
                    cont=FALSE;
               else
               {
                    r=hashfunc(hashtable[i].key);
                    a=(j<r && r<=i) || (r<=i && i<j) || (i<j &&
j<r);
               }
          }while(cont && a);
          if(cont)
               hashtable[j].key=hashtable[i].key;
     }while(cont);

   }

   //tac vu viewtable : xem chi tiet bang bam
   void viewtable()
   {
     int i;
     for(i=0;i<M;i++)
          printf("\nTable[%2d] : %4d\t",i,hashtable[i].key);
   }

   //chuong trinh chinh
   void main()
   {
     int i,n,p,q;
     int b, key, chucnang;
     char c;
     //khoi tao bang bam
     initialize();
     do
     {
          //menu chinh cua chuong tinh
          printf("\t\n\nCac chuc nang chinh cua chuong trinh :
\n");
          printf("\t1.Them nut moi vao bang bam\n");
          printf("\t2.Them ngau nhien mot nut vao bang bam\n");
          printf("\t3.Xoa nut tren bang bam\n");
          printf("\t4.Xoa toan bo bang bam\n");
          printf("\t5.Xem chi tiet bang bam\n");
          printf("\t6.Tim kiem tren bang bam\n");
```

```
          printf("\t0.Ket thuc chuong trinh\n");
          printf("\tChuc nang ban chon : ");
          scanf("%d",&chucnang);
          switch(chucnang)
          {
          case 1:
               {
                    printf("\nThem nut vao bang bam ");
                    printf("\nKhoa cua nut moi : ");
                    scanf("%d", &key);
                    insert(key);
                    break;
               }
          case 2:
               {
                    randomize();
                    printf("\nThem  ngau  nhien  nhieu  nut  vao
bang bam");
                    printf("\nBan muon them bao nhieu nut : ");
                    scanf("%d", &n);
                    for(i=0;i<n;i++)
                    {
                         key=random(1000);
                         insert(key);
                    }
                    break;
               }
          case 3:
               {
                    printf("\nXoa nut tren bang bam");
                    printf("\nkhoa cua nut can xoa : ");
                    scanf("%d", &key);
                    i=search(key);
                    if(i==M)
                         printf("Khong  co  nut  voi  khoa  can
xoa");
                    else
                    {
                         remove(i);
                         N--;
                    }
                    break;
               }
          case 4:
               {
                    printf("\nXoa toan bo bang bam");
                    printf("\nBan co chat khong (c/k) : ");
                    c=getch();
                    if(c=='c'||c=='C')
                         initialize();
                    break;
```

```
                }
            case 5:
                {
                    printf("\nXem chi tiet bang bam");
                    viewtable();
                    break;
                }
            case 6:
                {
                    printf("Tim kiem tren bang bam");
                    printf("Khoa can tim : ");
                    scanf("%d", &key);
                    if(search(key)==M)
                        printf("\nKhong tim thay");
                    else
                    {
                        printf("Tim thay tai dia chi %d trong
bang bam", search(key));
                        break;
                    }
                }
                scanf("%d",&key);
            }
        }while(chucnang!=0);
    }
```

3.5. TỔNG KẾT VỀ PHÉP BĂM

· Bảng băm đặt cơ sở trên mảng.

· Phạm vi các giá trị khóa thường lớn hơn kích thước của mảng.

· Một giá trị khóa được băm thành một chỉ mục của mảng bằng hàm băm.

· Việc băm một khóa vào vào một ô đã có dữ liệu trong mảng gọi là sự đụng độ.

· Sự đụng độ có thể được giải quyết bằng hai phương pháp chính: Phương pháp nối kết và phương pháp băm lại.

· Trong phương pháp băm lại, các mục dữ liệu được băm vào các ô đã có dữ liệu sẽ được đưa vào ô khác trong mảng.

· Trong phương pháp nối kết, mỗi phần tử trong mảng có một danh sách liên kết. Các mục dữ liệu được băm vào các ô sẽ được đưa vào danh sách ở ô đó.

Vấn đề Hàm băm

· Hàm băm dùng phương pháp chia: h(k) = k mod m

· m là kích thước bảng băm, k là khóa.

· Hàm băm dùng phương pháp nhân: h(k) = m(kA mod 1)

· Knuth đề nghị A = 0,6180339887

Số lần đụng đô: (ví dụ)

Kích thước bảng băm	PP chia	PP Nhân
200	698	699
512	470	466
997	309	288

1024	301	292

· Theo bảng trên kết quả cho thấy kích thước bảng băm tỷ lệ nghịch với số lần đụng độ. Số đụng độ còn phụ thuộc vào phương pháp sử dụng hàm băm.

· ***Hệ số tải*** *là tỉ số giữa các mục dữ liệu trong một bảng băm với kích thước của mảng.*

· *Hệ số tải cực đại trong phương pháp băm lại khoảng 0,5. Đối với băm kép ở Hệ số tải này (0,5), các phép tìm kiếm sẽ có chiều dài thăm dò tRung bình là 2.*

· Trong phương pháp băm lại , thời gian tìm kiếm sẽ là vô cực khi hệ số tải đạt đến 1.

· Điều quan trọng trong phương pháp băm lại là bảng băm không bao giờ được quá đầy.

· Phương pháp nối kết thích hợp với hệ số tải là 1.

· Với hệ số tải này, chiều dài thăm dò tRung bình là 1,5 khi phép tìm thành công, và là 2,0 khi phép tìm thất bại.

· Chiều dài thăm dò trong phương pháp nối kết tăng tuyến tính theo hệ số tải.

· Kích thước của bảng băm thường là số nguyên tố. Điều này đặc biệt quan trọng trong thăm dò bậc hai và trong phương pháp nối kết.

· Các bảng băm có thể dùng cách lưu trữ ngoại. Một cách để thực hiện việc này là cho các phần tử trong bảng băm chứa số lượng các khối của tập tin trên đĩa.

❖ **Bài tập củng cố:**

Cho dãy khóa các số nguyên có tối đa hai chữ số gồm:

30 19 5 25 22 35 23 21 20 12

a. Hàm băm được tính bằng tổng các chữ số của khóa chia cho 10 lấy phần dư (chiều dài mảng băm là M = 10). Ví dụ: khóa = 42, h(42)=(4+2) mod 10 = 6. Hãy bố trí lần lượt các khóa trên vào mảng băm với phương pháp băm tuyến tính.

b. Biết hàm băm thứ nhất ở câu a và hàm băm thứ hai $h_2(k)=1 + (a \bmod 5)$ với a bằng tổng các chữ số của khóa k. Hãy bố trí lần lượt các khóa trên vào mảng băm với phương pháp băm kép.

c. Cài đặt phương pháp băm tuyến tính và băm kép với hai hàm được cho ở trên.

CHƯƠNG 4
B-TREE VÀ BỘ NHỚ NGOÀI

❖ **Mục tiêu học tập:** Sau khi học xong chương này, người học có thể nắm vững và vận dụng:

- Phương pháp truy xuất dữ liệu trên bộ nhớ ngoài.
- Khái niệm B-Tree
- Các thao tác: thêm, xóa và tìm kiếm trên B-Tree.
- Khái niệm một số B-Tree cải tiến.
- Phương pháp tổ chức B-Tree ở bộ nhớ ngoài.

B-tree là một dạng của cây nhiều nhánh, B-tree đặc biệt hữu dụng đối với việc tổ chức dữ liệu ở bộ nhớ ngoài. Một node trong B-tree có thể có hàng chục thậm chí hàng trăm node con.

Trong chương này chúng ta sẽ xem xét cách tiếp cận đơn giản để tổ chức dữ liệu bên ngoài: thứ tự tuần tự, thảo luận về B-tree và vấn đề lưu trữ dữ liệu trên bộ nhớ ngoài.

4.1. TRUY XUẤT DỮ LIỆU TRÊN BỘ NHỚ NGOÀI

Các cấu trúc dữ liệu mà chúng ta đã thảo luận từ trước đến giờ đa số dựa trên lưu trữ dữ liệu trong bộ nhớ chính (thường gọi là RAM, viết tắt của Random Access Memory). Tuy nhiên, trong một vài tình huống, số lượng dữ liệu được xử lý là quá lớn để cùng một lúc đưa vào bộ nhớ chính. Trong trường hợp này các kiểu lưu trữ khác nhau là cần thiết. Thường các tập tin trên đĩa có dung lượng lớn hơn nhiều so với bộ nhớ chính; điều này là đúng bởi vì giá thành khá rẻ của chúng trên một đơn vị lưu trữ.

Tất nhiên, các tập tin trên đĩa cũng có thuận lợi khác: đó là khả năng lưu trữ lâu dài của chúng. Khi bạn tắt máy (hoặc nguồn điện bị hư), dữ liệu trong bộ nhớ chính sẽ bị mất. Các tập tin trên đĩa có thể lưu lại dữ liệu vô thời hạn khi nguồn điện bị tắt. Tuy nhiên, sự khác biệt chủ yếu là về kích cỡ mà chúng ta cần lưu ý ở đây.

Bất lợi của việc lưu trữ ngoài là sự truy xuất chậm hơn so với bộ nhớ chính. Sự khác biệt về tốc độ này có thể được giải quyết bằng các kỹ thuật khác nhau để làm tăng tính tính hiệu quả của chúng.

Một ví dụ trong việc lưu trữ ngoài đó là giả sử bạn viết một chương trình cơ sở dữ liệu để xử lý dữ liệu trong danh bạ điện thoại của một thành phố có kích thước tRung bình; cỡ 500,000 mục. Mỗi mục bao gồm tên, địa chỉ, số điện thoại, và các dữ liệu khác mà một công ty điện thoại thường sử dụng. Chúng ta giả sử rằng với lượng dữ liệu này là quá lớn so với bộ nhớ chính của một máy tính nào đó, nhưng sẽ trở nên rất nhỏ để lưu trữ trên ổ đĩa.

Kết quả là có một số lượng lớn dữ liệu trên đĩa. Làm thế nào cấu trúc nó để thực hiện các tính năng thông dụng mong muốn: tìm kiếm, chèn, xoá nhanh?

Có 2 vấn đề cần xem xét:

· Việc truy cập dữ liệu trên đĩa chậm hơn nhiều so với truy cập trên bộ nhớ chính.

· Thứ hai là việc truy cập nhiều mẫu tin cùng một lúc.

Truy xuất chậm

Bộ nhớ chính của máy tính làm việc với tín hiệu điện tử. Bất kỳ byte nào cũng có thể truy cập nhanh như các byte khác, với một phần nhỏ của micro giây (bằng 1/1,000,000 của giây).

Để truy cập một phần cụ thể dữ liệu trên ổ đĩa, đầu đọc/ghi trước tiên phải dịch chuyển đến rãnh phù hợp. Điều này được thực hiện bởi một mô tơ, hoặc thiết bị tương tự: đây là hoạt động cơ khí chiếm vài milli giây (bằng 1/1,000 của giây).

Kết quả là, thời gian truy cập ổ đĩa thường vào khoảng 10 milli giây. Điều này có nghĩa là chậm hơn 10,000 lần so với truy cập bộ nhớ chính.

Sự phát triển của kỹ thuật đã giảm thời gian truy xuất đĩa theo thời gian, nhưng thời gian truy xuất bộ nhớ chính giảm xuống nhanh hơn so với truy cập đĩa, vì thế sự chênh lệch giữa thời gian truy xuất bộ chính và ổ đĩa sẽ ngày càng lớn trong tương lai.

Truy xuất khối (block)

Một khi đã định vị đúng vị trí và tiến trình đọc (ghi) bắt đầu, ổ đĩa có thể chuyển một lượng lớn dữ liệu vào bộ nhớ chính một cách nhanh chóng và chính xác. Để làm được việc này và cũng nhằm đơn giản cơ chế điều khiển ổ đĩa, dữ liệu được lưu trữ trên đĩa thành các nhóm gọi là block, pages, allocation units, hoặc một vài tên gọi khác tuỳ thuộc vào hệ điều hành. ở đây chúng ta sẽ gọi chúng là khối (block).

Kích cỡ của khối biến đổi tuỳ thuộc vào từng hệ điều hành. Kích thước của ổ đĩa với các yếu tố khác, thường là luỹ thừa của 2. Đối với ví dụ về danh bạ điện thoại như nêu ở trước, giả sử một khối có kích thước là 8,192 byte (2^{13}). Kết quả là cơ sở dữ liệu danh bạ điện thoại sẽ cần 256,000,000 byte chia cho 8,192 byte trên một khối, nghĩa là sẽ có 31,250 khối.

Chương trình sẽ hiệu quả khi nó yêu cầu thao tác đọc hoặc ghi với kích thước là bội số của kích thước khối. Nếu muốn đọc 100 byte, hệ thống sẽ đọc một khối 8,192 byte và chỉ lấy 100 byte, số còn lại sẽ không dùng đến. Hoặc nếu muốn đọc là 8,200 byte, hệ thống sẽ đọc 2 khối hay 16,384 byte nhưng chỉ lấy hơn một nửa của số byte này (8,200). Vì thế bạn phải tổ chức chương trình sao cho tại mỗi thời điểm nó chỉ làm việc trên một khối dữ liệu, điều này sẽ làm tối ưu sự truy xuất.

Giả sử kích thước mỗi mẫu tin của danh bạ điện thoại là 512 byte, bạn có thể lưu 16 mẫu tin thành một khối (8,192 chia cho 512), như trình bày ở hình 4.1. Vì thế, để tính hiệu quả đạt đến mức tối đa bạn phải đọc 16 mẫu tin tại mỗi thời điểm (hoặc là một bội số của 16).

Kích thước mỗi mẫu tin thường là bội số của 2. Điều này làm cho số lượng toàn bộ của chúng sẽ luôn luôn vừa với một khối.

Kích thước trình bày trong ví dụ danh bạ điện thoại của mẫu tin, khối,... chỉ là minh họa; Chúng sẽ biến đổi phụ thuộc vào số lượng và kích thước của mẫu tin và các ràng buộc về phần cứng và phần mềm khác. Khối thường chứa đựng hàng trăm mẫu tin, và các mẫu tin này có thể lớn hơn hoặc nhỏ hơn 512 byte.

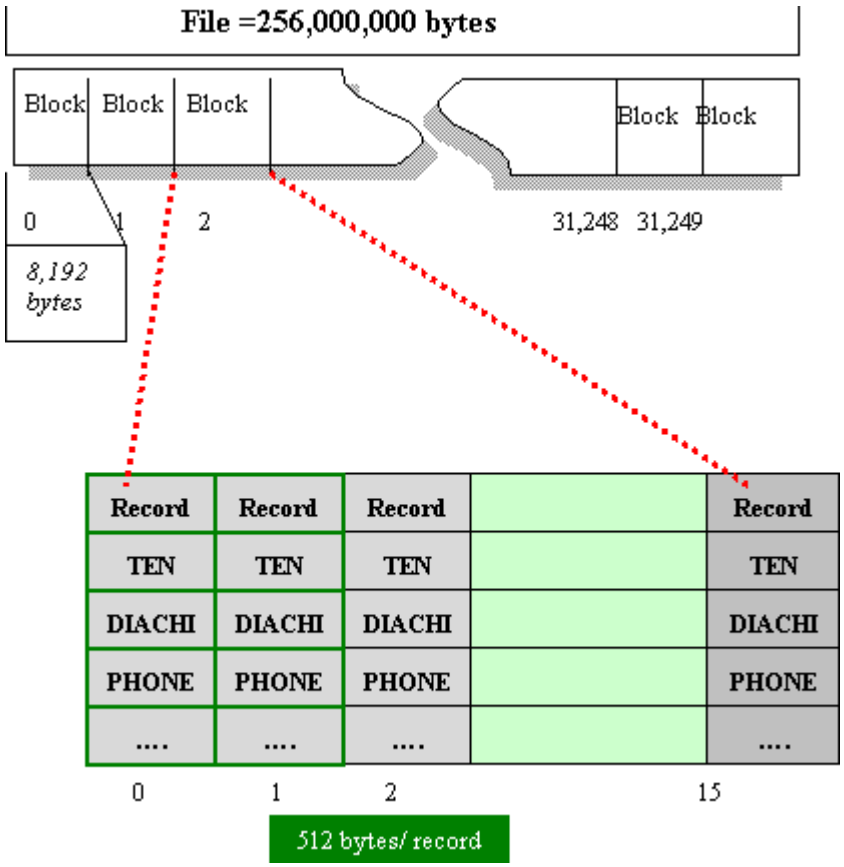
Một khi đầu đọc/ghi định vị đúng vị trí (như đã trình bày ở trên), việc đọc một khối rất nhanh, chỉ tốn vài milli giây. Vì thế, việc truy cập ổ đĩa để đọc hoặc ghi mỗi khối sẽ không phụ thuộc vào kích thước của khối. Điều này có nghĩa là khối càng lớn thì càng có hiệu quả khi bạn đọc hoặc viết một mẫu tin đơn (giả sử bạn sử dụng tất cả các mẫu tin trong khối).

Thứ tự tuần tự

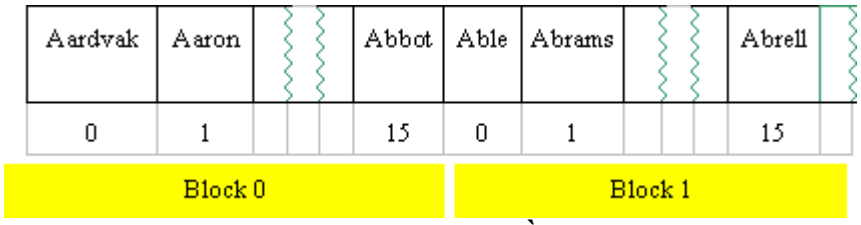
Có một cách để sắp xếp dữ liệu danh bạ điện thoại của tập tin trên đĩa đó là sắp xếp tất cả mẫu tin theo một vài khóa nào đó, giả sử sắp theo thứ tự alpha của họ. Nếu vậy thì mẫu tin của Joseph Aardvark là mẫu tin đầu tiên, v.v. Điều này như trình bày trong hình 4.12.

Tìm kiếm

Để tìm kiếm trên một tập tin có thứ tự tuần tự theo họ, cụ thể là Smith, bạn có thể sử dụng thuật toán tìm kiếm nhị phân. Bạn bắt đầu bằng việc đọc 1 khối các mẫu tin từ chính giữa của tập tin. 16 mẫu tin trong khối được đọc cùng một lúc vào một vùng đệm có kích thước 8,192 byte của bộ nhớ chính.



Hình 4.1 Khối và mẫu tin



Hình 4.2 Thứ tự tuần tự

Chúng ta đã biết, tìm kiếm nhị phân trong bộ nhớ chính sẽ cần khoảng $\log_2 N$ lần so sánh, với 500,000 mục sẽ cần khoảng 19 lần. Nếu mỗi lần so sánh, giả sử chiếm khoảng 10 micro giây thì việc tìm kiếm chiếm trên 190 micro giây, hay 2/10,000 giây.

Tuy nhiên, chúng ta đang xử lý dữ liệu lưu trên đĩa. Bởi vì mỗi lần truy cập đĩa là quá tốn thời gian, nên sẽ rất quan trọng để tập tRung vào giải quyết câu hỏi là cần thiết phải truy cập ổ đĩa bao nhiêu lần hơn là câu hỏi có bao nhiêu mẫu tin lưu trên đó. Thời gian để đọc một khối các mẫu tin sẽ lớn hơn nhiều so với thời gian để tìm kiếm trên 16 mẫu tin của 1 khối trong bộ nhớ chính.

Việc truy xuất đĩa chậm hơn nhiều so với truy cập trên bộ nhớ, nhưng tại một thời điểm chúng ta truy cập một khối, và có một vài khối xa hơn các mẫu tin. Trong ví dụ của chúng ta có 31,250 khối. Log_2 của số này vào khoảng 15, vì vậy theo lý thuyết chúng ta cần thiết phải truy cập đĩa 15 lần để tìm kiếm mẫu tin ta muốn.

Trong thực tế con số này có thể giảm xuống bởi vì chúng ta đọc 16 mẫu tin một lần. Trong giai đoạn đầu của việc tìm kiếm nhị phân nó không giúp cho ta có nhiều mẫu tin trong bộ nhớ bởi vì việc truy cập kế tiếp ở đoạn xa của tập tin. Tuy nhiên, khi chúng ta tìm gần tới mẫu tin mong muốn, mẫu tin kế tiếp mà chúng ta cần có thể đã nằm trong bộ nhớ bởi vì nó là một phần nằm trên khối gồm 16 mẫu tin. Điều này có thể giảm số lần so sánh xuống 2 lần

hoặc nhiều hơn nữa lần. Kết quả chúng ta cần khoảng 13 lần truy cập đĩa (15-2), khi 10 milli giây trên một lần truy cập thì chiếm khoảng 130 milli giây, hay 1/7 giây. Điều này chậm hơn nhiều so với việc truy cập trong bộ nhớ, nhưng nó không quá tồi.

Thao tác thêm vào và loại bỏ

Việc thêm vào và loại bỏ một mục dữ liệu từ tập tin có thứ tự tuần tự không hiệu quả. Vì dữ liệu là có thứ tự, cả 2 thao tác trên đều yêu cầu dịch chuyển tRung bình khoảng một nữa các mẫu tin, và vì thế sẽ dẫn đến dịch chuyển một nữa các khối.

Dịch chuyển mỗi khối đòi hỏi cần phải có 2 lần truy cập đĩa, 1 lần dùng để đọc và một lần dùng để ghi. Một khi điểm cần chèn được tìm thấy, khối chứa đựng điểm này sẽ được đọc vào buffer bộ nhớ. Mẫu tin trước đó của khối được lưu lại, và một lượng các mẫu tin thích hợp sẽ được đẩy lên để nhường chỗ cho mẫu tin mới cần chèn vào. Sau đó, nội dung của vùng đệm sẽ được ghi lại đĩa.

Kế đó, khối thứ 2 được đọc vào vùng đệm. Mẫu tin trước đó của nó cũng lưu lại, tất cả các mẫu tin khác được đẩy lên, và mẫu tin ở khối trước đó được chèn vào phần đầu của vùng đệm. Sau đó, nội dung của vùng đệm được ghi trở lại đĩa. Tiến trình này tiếp tục cho đến khi tất cả các khối mà vượt quá điểm chèn được ghi lại hết.

Giả sử rằng có 31,250 khối, chúng ta phải đọc và ghi chúng (tRung bình khoảng) 15,625 lần, mà mỗi lần đọc và ghi chiếm khoảng 10 mili giây, do đó sẽ đòi hỏi nhiều hơn 5 phút để thực hiện việc chèn một đầu vào đơn giản. Điều này là không thể thực thi nếu có hàng ngàn các tên để thêm vào danh bạ điện thoại.

Một rắc rối khác nữa đối với thứ tự tuần tự đó là nó chỉ làm việc nhanh với một khóa. Tập tin của chúng ta được sắp xếp bởi tên. Nhưng giả sử rằng muốn tìm kiếm một số điện thoại cụ thể, sẽ không thể sử dụng được thuật toán tìm kiếm nhị phân, bởi vì dữ liệu được sắp xếp bởi trường tên. Điều này dẫn đến phải duyệt toàn thể tập tin, từng khối một, sử dụng truy cập tuần tự. Điều này yêu cầu đọc đĩa tRung bình khoảng một nữa các khối, chiếm khoảng 2.5 phút, hiệu suất này quả thật không thể chấp nhận được đối với một sự tìm kiếm đơn giản. Cần thiết phải có phương pháp tổ chức lưu dữ liệu trên đĩa hiệu quả hơn.

4.2. B-TREE

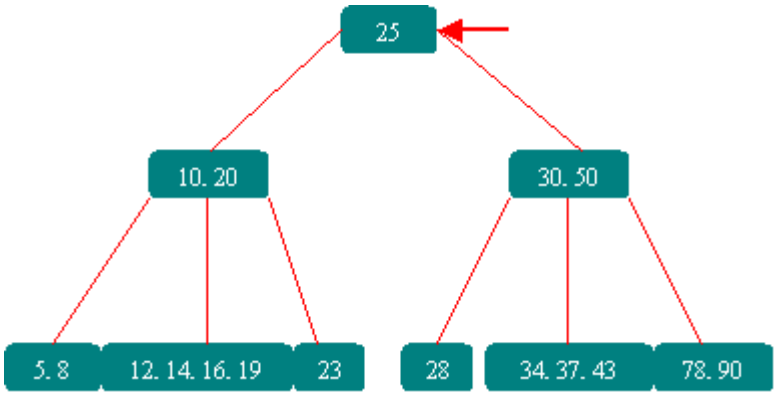
B-tree là cấu trúc dữ liệu phù hợp cho việc lưu trữ ngoài do R.Bayer và E.M.McCreight đưa ra năm 1972. Bên trong mỗi nút của B-cây, dữ liệu được xếp thứ tự một cách tuần tự bởi khoá. Bậc của B-tree là số các node con mà mỗi node có thể có.

4.2.1. Định nghĩa B-Tree:

Một B-tree bậc n có các đặc tính sau:

- Mỗi node có tối đa 2*n khoá.
- Mỗi node (không là node gốc) có ít nhất là n khoá.
- Mỗi node hoặc là node lá hoặc có m+1 node con (m là số khoá của node này)

Ví dụ:



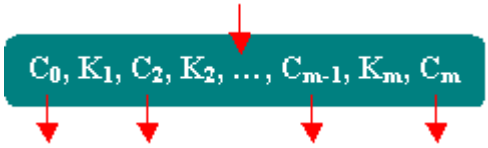
Hình 4.3. B-tree bậc 2 có 3 mức

Khai báo:

```
typedef struct NodeType
{
    int numtree; // số cây con của node hiện hành
    int Key[Order]; // mảng lưu trữ 3 khoá của node
    int Branch[Order]; // các con trỏ chỉ đến các node con
} NodeType;
typedef struct Nodetype *NODEPTR; // con trỏ node
NODEPTR Root; // con tro node goc
```

4.2.2. Các phép toán trên B-Tree

Tìm kiếm



Hình 4.4

Xét node trong hình 4.4, khoá cần tìm là X. Giả sử nội dung của node nằm trong bộ nhớ. Với m đủ lớn ta sử dụng phương pháp tìm kiếm nhị phân, nếu m nhỏ ta sử dụng phương pháp tìm kiếm tuần tự. Nếu X không tìm thấy sẽ có 3 trường hợp sau xảy ra:

i) $K_i < X < K_{i+1}$. Tiếp tục tìm kiếm trên cây con C_i

ii) $K_m < X$. Tiếp tục tìm kiếm trên C_m

iii) $X < K_1$. tiếp tục tìm kiếm trên C_0

Quá trình này tiếp tục cho đến khi node đúng được tìm thấy. Nếu đã đi đến node lá mà vẫn không tìm thấy khoá, việc tìm kiếm là thất bại.

Phép toán NODESEARCH

Trả về vị trí nhỏ nhất của khóa trong nút p bắt đầu lớn hơn hay bằng k. Trường hợp k lớn hơn tất cả các khóa trong nút p thì trả về vị trí p-> numtrees-1

```
int nodesearch (NODEPTR p, int k)
{
    int i;
    for(i=0; i< p->numtrees -1 && p->key[i] < k; i++);
    return (i);
}
```

Phép toán nodesearch được dùng để tìm khóa k có trong nút p hay không. Nếu khóa k không có trong nút p thì phép toán này trả về vị trí giúp chúng ta chọn nút con phù hợp của p để tiếp tục tìm khóa k trong nút con này.

Phép toán SEARCH:

Tìm khóa k trên B-Tree. Con trỏ p xuất phát từ gốc và đi xuống các nhánh cây con phù hợp để tìm khóa k có trong một nút p hay không

Nếu có khóa k tại nút p trên cây:

- Biến found trả về giá trị TRUE
- Hàm search() trả về con trỏ chỉ nút p có chứa khóa k
- Biến position trả về vị trí của khóa k có trong nút p này

Nếu không có khóa k trên cây: lúc này p=NULL và q(nút cha của p) chỉ nút lá có thể thêm khóa k vào nút này được.

- Biến found trả về giá trị FALSE
- Hàm search() trả về con trỏ q là nút lá có thêm nút k vào
- Biến position trả về vị trí có thể chèn khóa k vào nút lá q này

```
NODEPTR search(int k, int *pposition, int *pfound)
{
  int i;
  NODEPTR p, q;
  q = NULL;
  p = Root;
  while (p !=NULL)
  {
    i = nodesearch (p, k);
    if(i< p->numtree-1 && k == p->key[i]) //tim thay
    {
       *pfound = TRUE;
       *pposition = i; // vi trí tìm thay khoa k
       return(p); // node co chua khoa k
     }
    q = p;
    p = p ->Branch[i];
    }
/*Khi thoat khoi vong lap tren la khong tim thay, luc nay
p=NULL, q la node la co the them khoa k vao node nay, position
la vi tri co the chen khoa k*/
    *pfound = FALSE;
    *pposition = i;
    return (q); //tra ve node la
}
```

Phép Duyệt:

Duyệt các khóa của B-Tree theo thứ tự từ nhỏ đến lớn-bằng phương pháp đệ qui

```
void traverse(NODEPTR proot)
{
  int i;
  if(proot == NULL)   //dieu kien dung
       return;
  else      // de qui
  {
/* vong lap duyet nhanh cay con Branch[i] va in khoa key[i]
cua node proo*/
  for(i = 0; i < proot -> numtress-1; i++)
       {
            traverse (proot ->Branch[i]);
            printf ("%8d", proot -> key[i]);
}
//duyet nhanh cay con cuoi cung cua node proot
traverse (proot -> Branch[proot -> numtrees-1]);
}
}
```

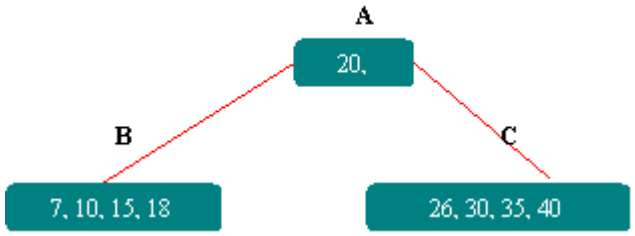
Thêm vào :

Trước khi đưa ra giải thuật thêm một phần tử mới vào B-Tree, ta xem tình huống cụ thể qua các ví dụ sau :

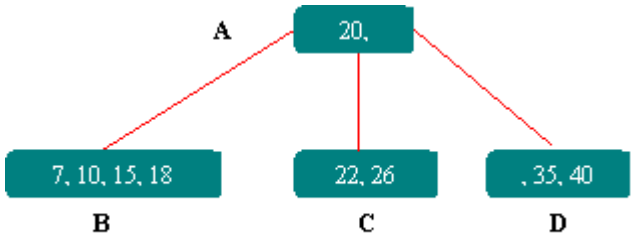
Ví dụ 1:

- Thêm x=22 vào B-Tree ở hình 4.5a. Khóa 22 chưa có trong cây. Nhưng không thể thêm vào node C vì node C đã đầy.

- Do đó tách node C thành hai node : node mới D được cấp phát và m+1 khóa được chia đều cho 2 node C và D, và khóa ở giữa được chuyển lên node cha A : Hình 4.5b



Hình 4.5 a



Hình 4.5 b

Như vậy, việc thêm một khóa mới vào B-Tree có thể gây ra việc tách node và việc tách node có thể lan truyền ngược lên node cha, trong trường hợp đặc biệt lan truyền đến tận gốc của B-Tree

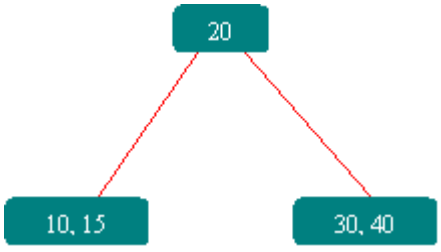
Ví dụ 2 : Xem quá trình tạo B-Tree từ dãy các khóa sau :

20; 40 10 30 15; 35 7 26 18 22; 5; 4 13 46 27 8 32; 38 24 45 25

Sau khi thêm vào khóa 30 :

Hình 4.6 a

Khi thêm vào 15 thì node này bị đầy, do đó trường hợp này tạo thành 2 node mới : phần tử ở giữa là 20 bị đẩy lên tạo thành một node mới, các phần tử còn lại chia cho 2 node : node cũ chứa 10, 15 và node mới thứ 2 chứa 30,40

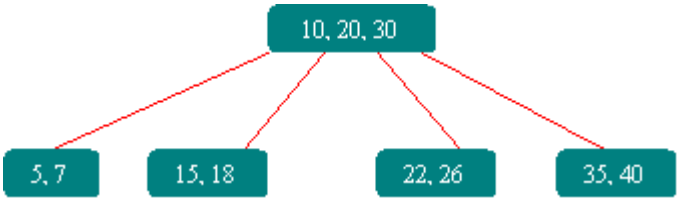


Hình 4.6 b

Thêm vào các khóa 35, 7,26 và 18. Đến khi thêm khóa 22 cũng có sự đầy node dẫn đến việc tách node:

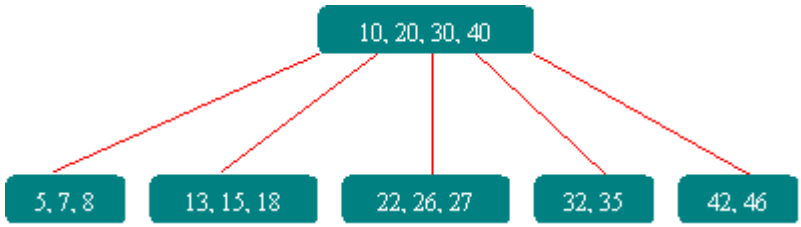
Hình 4.6 c

Thêm vào 5 cũng có sự đầy node (node đang chứa 4 khóa 7, 10, 15, 18) dẫn đến việc tách node :



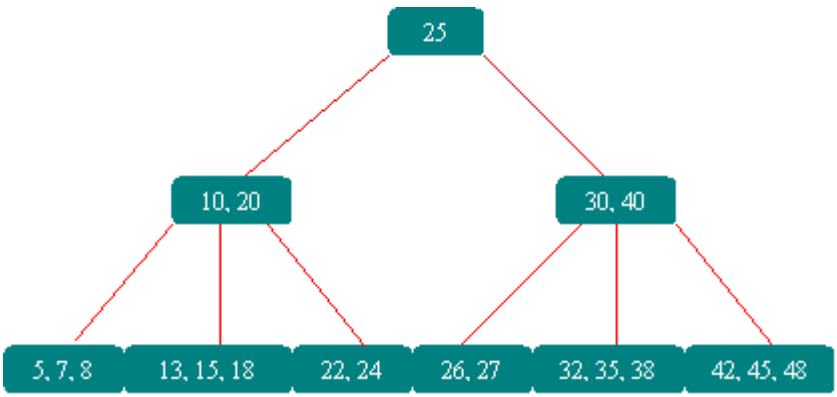
Hình 4.6 d

Thêm vào các khóa 42, 13, 46, 27 và 8. Đến khi thêm 32 có sự tách node :



Hình 4.6 e

Thêm vào 38, 24 va 45. Thêm 25 vào có sự tách node và cho thấy sự lan truyền tách node ngược lên về phía gốc : 25 thêm vào node (22, 24 26, 27) làm node này bị tách và 25 được đưa lên node cha (10, 20, 30, 40) làm node này bị tách thành 2 node và khoá 25 được đưa lên thành node gốc mới



Hình 4.6 f

Phép toán INSERT

Thêm khóa k vào vị trí position của nút lá s (s và position do phép toán search() trả về)
· Nếu nút lá s chưa đầy: gọi phép toán insnode để chèn khóa k vào nút s
· Nếu nút lá s đã đầy: tách nút lá này thành 2 nút nửa trái và nửa phải

```
void insert (NODEPTR s, int k, int position)
{
    NODEPRT nd, nd2, f, newnode;
    int pos, newkey, midkey;
    //khoi dong cac tri truoc khi vao tron vong lap tach
cac node day nd
    nd = s;
    newkey = k;
    newnode = NULL; // vi nd la node la nen gan newnode la
NULL
    pos = position;
    f = father (nd);
    // Vong lap tach cac node day nd
    while (f != NULL && nd -> nemtrees == ODER)
    {
      split(nd, newkey, newnode, pos, &nd2, &midkey);
      // Gan lai cac tri sau lan tach node truoc
      nd = f;
      newkey = midkey;
      newnode = nd2;
      pos = nedesearch (f, midkey);
      f = father (nd);
    }
    // Truong hop node nd chua day va nd khong phai la node
goc
    if(nd - > numtrees < ORDER)
    {
       //chen newkey va newnode tai vi tri pos cua node nd
       insnode (nd, newkey, newnode, pos);
       return;
    }
    //Truong hop node nd la node goc bi day, tach node goc
nay va tao node goc moi
    split (nd, newkey, newnode, pos, &nd2, &midkey);
    Root = makeroot (midkey); // tao node goc moi
    // Gan lai hai nhanh cay con cua node goc moi la nd va
nd2
    Root -> Branch[0] = nd;
    Root -> Branch[1] = nd2;
    }
```

Khi thêm một khóa vào B-Tree chúng ta có thể viết như sau:

```
printf("\n Noi dung khoa moi: ");
scanf("%d", &k);
// truong hop B-Tree bi rong khi tao node goc
if(Root == NULL)
  Root = makeroot(k);
else
{
```

```
      s = search(k, &pos, &timthay);
      if(timthay)
           printf("Bi tRung khoa, khong them khoa %d vao B-Tree
        duoc", k);
      else
      insert (s, k, pos);
    }
```

Phép toán SPLIT:

Tách node đầy nd, phép toán này được gọi bởi phép toán I**NSERT**

· nd là nút đầy bị tách, sau khi tách xong nút nd chỉ còn lại một nửa số khóa bên trái

· newkey, newnode và pos là khóa mới, nhánh cây con và vị trí chèn vào nút nd

· Nút nd2 là nút nửa phải có được sau lần tách, nút nd2 chiếm một nửa số khóa bên phải

· Midkey là khó ngay chính giữa sẻ d9uoc757 chèn vào nút cha

```
    void split (NODEPTR nd, int newkey, NODEPTR newnode, int
pos, NODEPTR *pnd2, int *pmidkey)
    {
    NODEPTR p;
      P = getnode(); //cap phat node nua phai
      /*truong hop chen newkey va newnode vao node      nua
phai*/
      if(pos > Ndiv 2)
      {
    copy(nd, Ndiv 2+1, ORDER - 2, p);
      insnode (p, newkey, newnode, pos-Ndiv 2 -1);
      nd->numtrees = Ndiv 2+1; /*so nhanh cay con con   lai  cua
node nua trai*/
      *pmidkey = nd -> key[Ndiv2];
    *pnd2 = p;
      return;
    }
    // truong hop newkey la midkey
    if(pos == Ndiv2)
    {
      copy(nd, Ndiv2, ORDER-2, p);
    nd->numtrees = Ndiv 2+1; /*so nhanh cay con con    lai  cua
node nua trai*/
      /*Dieu chinh lai node con dau tien cua node nua   phai*/
      p -> Branch[0] = newnode;
      *pmidkey = nd -> key[Ndiv2];
      *pnd2 = p;
      return;
    }
    /* Truong hop chen newkey va newnode vao node nua trai*/
    if(pos < Ndiv2)
    {
        copy(nd, Ndiv2, ORDER-2, p);
        nd->numtrees = Ndiv 2+1; /*so nhanh cay con con lai cua
    node nua trai*/
        *pmidkey = nd -> key[Ndiv2 - 1];
        insnode(nd, newkey, newnode, pos);
        *pnd2 = p;
```

```
        return;
    }
    }
```

Phép toán INSNODE

Chèn khóa newkey vào vị trí pos của nút chưa đầy p,và chèn nhánh cây con newnode vào vị trí bên phải cuả khóa newkey

```
    void insnode (NODEPTR p, int newkey, NODEPTR newnode, int
pos)
    {
        int i;
        /*doi cac nhanh cay con va cac khoa tu vi tri  pos  tro
    ve sau xuong mot vi tri*/
        for(i = p->numtress - 1; i >= pos+1; i--)
        {
           p -> Branch[i+1] = p -> Branch[i];
           p -> key[i] = p -> key[i - 1];
        }
        // gan khoa newkey vao vi tri pos
        p -> key[pos] = newkey;
        // Gan nhanh newnode la nhanh cay con ben phai cua khoa
    newkey
        p -> Branch[pos + 1] = newnode;
        //tang so nhanh cay con cua node p len 1
        p -> numtrees +=1;
    }
```

***Phép toán* COPY**

Chép các khóa (và nhánh cây con) từ vị trí first đến vị trí last của nút nd (nút nửa trái) sang nút nd2 (nửa nút phải) . Phép toán này được gọi bởi phép toán split

```
    void copy(NODEPTR nd, int first, int last, NODPTR nd2)
    {
      int i;
      // copy cac khoa tu node nd qua node nd2
      for(i = first; i < last, i++)
           nd2 -> key[i-first] = nd -> key[i];
    // copy caùc nhanh cay con tu node nd qu nd2
      for(i = first; i < last+1, i++)
      nd2 -> con[i-first] = nd -> Branch[i];
      nd2 ->numtrees = last - first +2 // so nhanh cay con cua
node nd2
    }
```

4.3. B-TREE CẢI TIẾN :

Vì tất cả các nút (trừ nút góc) của B-Tree đều đầy hơn một nửa nên cấu trúc B-Tree khá tối ưu bộ nhớ- chiếm dụng bộ nhớ 50% hay hơn. Để dùng bộ nhớ hiệu quản hơn người ta cải tiến B-Tree thành những cấu trúc sau :

4.3.1. B*-Tree

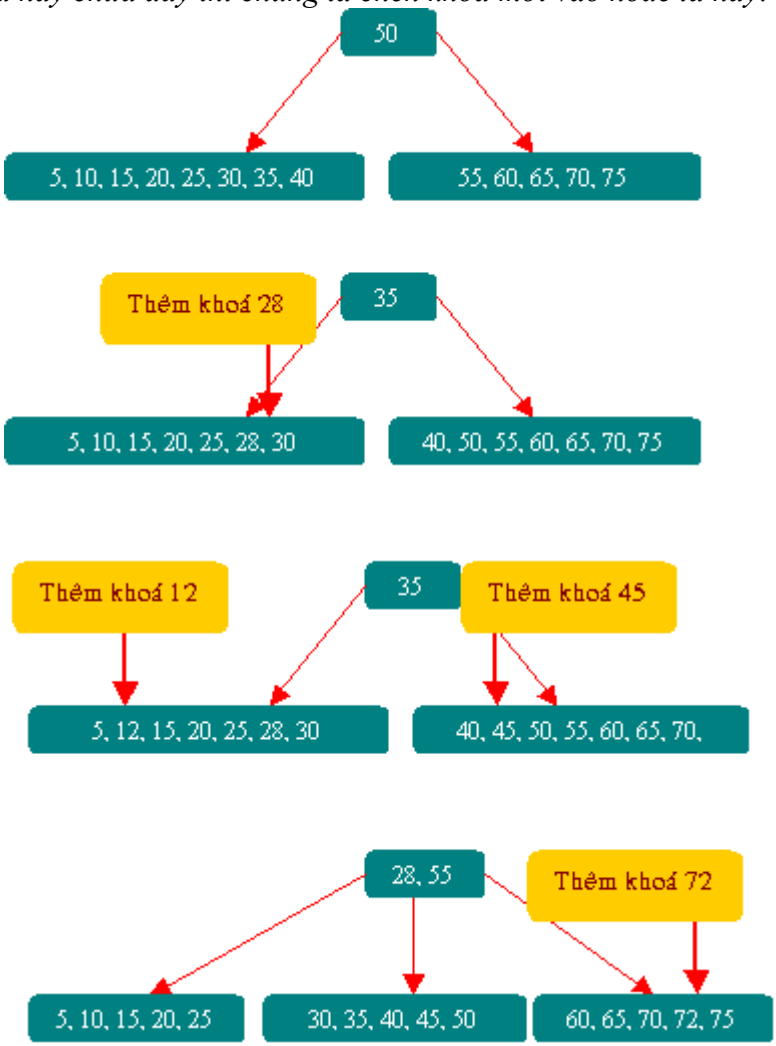
B*-Tree bậc ORDER cũng là B-Tree bậc ORDER nhưng tất cả các nút trên cây (trừ nút gốc) phải đầy hơn 2/3.

Chúng ta thấy cấu trúc B*Tree tối ưu bộ nhớ hơn B-Tree, bộ nhớ hiệu dụng đạt được 67% hay hơn.

Khi thêm một khóa vào B*-Tree chúng ta phải tìm nút lá phù hợp để chèn khóa mới vào

nút lá này :

Nếu *node lá này chưa đầy thì chúng ta chèn khoá mới vào node là này:*

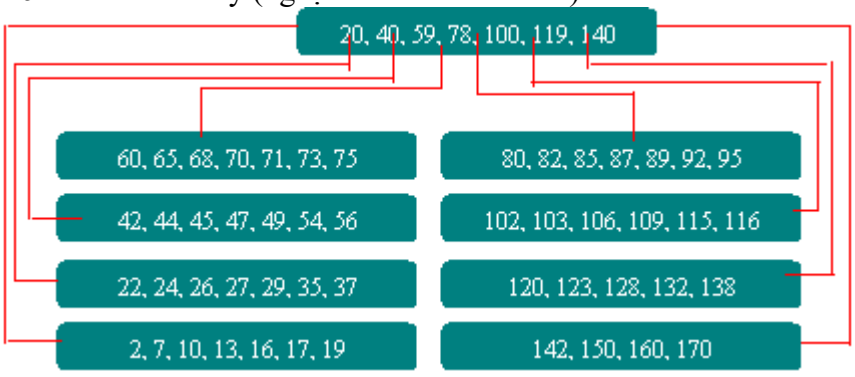


***Hình 4.7** - Thêm vào B*Tree*

Nếu nút lá này đã đầy chúng ta phân bố lại các khóa giữa ba nút : hai nút anh em kế nhau và nút cha. Trường hợp nút lá anh em cũng đã đầy chúng ta có thể cấp phát một nút anh em mới và phân bố các khóa giữa ba nút anh em và nút cha.

4.3.2. Compact B-Tree

Là B-Tree mà các nút đều đầy (ngoại trừ vài nút lá cuối).



***Hình 4.8** - Một Compact B-Tree bậc 8*

Ưu điểm của Compact B-Tree :

· Tìm kiếm một khóa trên cây nhanh .

· Là cấu trúc sử dụng bộ nhớ trong 100%.

Khuyết điểm của Compact B-Tree :

· Giải thuật thêm một khóa vào Compact B-Tree rất phức tạp, chi phí để chuyển cây về dạng Compact B-Tree rất lớn

Nếu dùng thực hiện phép toán insert của B-Tree trên Compact B-Tree thì quá trình tách nút sẽ diễn ra đồng loạt từ nút lá đến nút gốc !

Cấu trúc Compact B-Tree chỉ phù hợp khi cây ít bị thay đổi (ít thực hiện phép toán thêm khóa, xóa khóa).

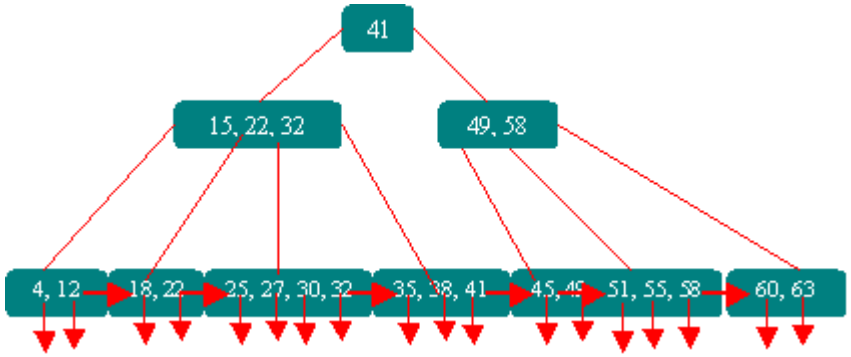
Người ta thường nên B-Tree (chuyển B-Tree về dạng Compact B-Tree trong thời gian hệ thống rãnh rỗi, chẳng hạn nén vào lúc trời khuya thanh vắng…)

4.3.3. B+ -Tree

Vì B-Tree duyệt cây phức tạp người ta cải tiến thành B+Tree để giúp duyệt cây hiệu quản hơn.

B+ Tree là B-Tree (lúc này cho phép trùng khóa) mà tất cả các khóa trên cây đều có mặt ở các nút lá. Các nút lá được liên kết nhau từ trái qua phải hình thành một danh sách liên kết giúp duyệt các khóa trên cây.

Các khóa ở nút là liên kết với con trỏ chỉ mẫu tin của khóa khóa lưu ở bộ nhớ ngoài.



Hình 4.9 - Cây B+Tree bậc 4

Các phép toán trên B+ -Tree

Truy xuất mẫu tin có khóa k :

Cũng như search của B-Tree, nhưng quá trình tìm kiếm chỉ kết thúc khi đi đến lá để truy xuất được con trỏ chỉ mẫu tin của khóa ở bộ nhớ ngoài.

Duyệt B+ Tree:

Chúng ta có thể duyệt toàn bộ cây hay duyệt từ khóa……..đến khóa……..bằng cách duyệt danh sách liên kết của các nút lá.

Thêm khóa :

Tương tự B-Tree, tuy nhiên khi tách nút khóa giữa midkey được lưu ở nút nửa trái và đồng thời lưu ở nút cha.

4.4. B-TREE VÀ BỘ NHỚ NGOÀI

Giới thiệu :

Để tiết kiệm bộ nhớ trong người ta thường tổ chức B-Tree ở bộ nhớ ngoài (đĩa từ, băng từ,…) khi truy xuấ B-Tree người ta chỉ nạp một phần nhỏ của B-Tree vào bộ nhớ trong để xử lý.

Tất cả các nút trên B-Tree lúc này được lưu trữ ở bộ nhớ ngoài, mỗi nút cũng chứa các thông tin như : số nhánh cây con của nút (numtrees), các khóa có trong nút (key[]), các địa chỉ của các nút Branch (Branch []).

Ngoài ra chúng ta cần biết những thông tin để quản lý B-Tree ở bộ nhớ ngòai như địa chỉ của nút gốc, số nút hiện có trên B-Tree,….Những thông tin này có thể lưu ở bộ nhớ ngòai hay được xác định trong chương trình xử lý

Vấn đề tìm kiếm một khóa trên B-Tree tổ chức ở bộ nhớ ngòai :

Khi tìm kiếm một khóa trên B-Tree chúng ta nạp vào bộ nhớ trong các nút trên đường đi phù hợp từ nút gốc đến nút lá (nạp vào mảng chứa các nút pathnode), và tiến hành tìm kiếm khóa trên các nút đã nạp vào bộ nhớ trong này.

Khi không tìm thấy khóa trên B-Tree phép toán search đã nạp vào bộ nhớ trong các nút trên đường phù hợp từ nút gốc đến nút lá (mảng pathnode[]) giúp chúng ta dùng mảng pathnode[]) này để xử lý việc thêm khóa vào B-Tree.

Vấn đề thêm một khóa trên B-Tree tổ chức ở bộ nhớ ngoài :

Khi thêm một khóa vào B-Tree trước tiên chúng ta gọi phép toán search để kiểm tra có bị trùng khóa không. Nếu không bị trùngkhóa thì phép toán search đã nạp vàp mảng pathnode[] chứa các nút trên đường đi phù hợp từ nút gốc đến nút lá, chúng ta dùng mảng pathnode[] này để xử lý việc thêm khóa vào B-Tree như mô tả sau :

Nếu nút cuối của pathnode[] (cũng là nút lá của B-Tree cần thêm khóa vào) chưa đầy thì chúng ta chèn khóa mới vào nút này, sau đó cập nhật lại nội dung của nút lá này ở bộ nhớ ngòai.

Nếu nút cuối của pathnode[] đã đầy (nút này đã có ORDER-1 khóa và ORDER nhánh cây con), chúng ta tách nút này thành hai nút : nút nửa trái (gọi là nút nd) và nút nửa phải (gọi là nút nd2). Chúng ta phải cập nhật lại nút nửa trái (vì nút chỉ còn lại một nửa) và tạo mới nút nửa phải cho B-Tree ở bộ nhớ ngoài.

Khóa chính giữa (midkey) và nút con nd2 được chèn vào nút cha là nút ngay trước trong pathnode[]. Vấn đề được xử lý tương tư khi chèn khóa midkey và nút con nd2 vào nút cha

…..

Giới thiệu các phép toán trên B-Tree tổ chức ở bộ nhớ ngoài :

Chúng ta phải hiệu chỉnh, thêm/bớt một số phép toán khi tổ chức B-Tree ở bộ nhớ ngoài.

Gọi :

pathnode[] là mảng chứa các nút trên đường đi (path) từ nút gốc đến nút lá trên B-Tree.

pathnode[i] là nút thứ i trong pathnode[].

Location là mảng chứa các địa chỉ của các nút trong pathnode[].

Location [i] là địa chỉ của nút i trong pathnode[].

Node (loc) là nút trên B-Tree tại địa chỉ loc (địa chỉ Location).

Các khai báo :

```
// Cac truc mot node B-Tree o bo nho ngoai
struct node
{
  int numtrees ; // so nhanh cay con cua node
  ont key [ORDER-1] ; // cac khoa cua node
int Branch [ORDER] ; /*cac dia chi cua cac node con cua node gia su dia chi la so nguyen */
} ;
struct node pathnode [20];/*mang chua cac node tren duong di tu node goc den node la */
int location [20] ; /* mang chua dia chi cua cac node tren pathnode */
```

Phép toán makeroot (x) (gọi : ptee = makeroot (x))

Tạo một nút gốc mới x cho B-Tree. Phép toán này cấp phát một block bộ nhớ ngòai cho nút x và trả về địa chỉ của nút gốc mới (Root) được cấp phát

Node (addr) = pathnode[i] ;

Phép toán access (i,loc) :

Phép toán này truy xuất nút ở địa chỉ loc chép vào pathnode[i], đồng thời chép địa chỉ loc vào location [i] :

```
pathnode[i] = node (loc);
location [i] = loc ;
```

Phép toán replace (i, loc) :

Khi nút pathnode[i] bị thay đổi chúng ta gọi phép toán này để cập nhật lại nút pathnode[i] tại địa chỉ loc của B-Tree ở bộ nhớ ngòai

Node (location [i] = pathnode[i])

Phép toán search :

Tìm khóa k trên B-Tree ở bộ nhớ ngoài

Nếu có :

- Biến found trả về giá trị TRUE.
- Hàm search () trả về nút vị trí j trong pathnode có chứa khóa k.
- Biến position trả về vị trí của khóa k có trên nút này.

Nếu không có :

- Biến found trả về giá trị FALSE.
- Hàm search () trả về nút vị trí cuối cùng pathnode là nút lá có thể thêm khóa k vào.
- Biến position trả về vị trí của khóa k nếu thêm nó vào nút này.

```
int search (int k, int *pposition, int *pfound)
{
        int p, i, j ;
        p = Root ;/* p xuat phat tu node goc cua B-Tree*/
        j = -1 ;  // j la chi so tren pathnode
        while (p ! = -1)
        {
        j++ ;
        access(j,p);//nap node p vao pathnode [j]
        i = nodesearch (j, k); /* tim khoa k trong pathnode
[j] */
        if (i < pathnode [j]. numtrees-1 &&
        k == pathnode [j]. key [i])
        {
             * pfound = TRUE ;
             * pposition = i ;
             return j ;
        }
  p = pathnode [j].Branch [i];/* p di xuong node con*/
    }
    /* truong hop khong co khoa k tren B-Tree, luc nay da hinh
thanh mang pathnode [] chua cac  node tren duong di tu node goc
den node la */
    *pfound = FALSE ;
    * ppoosition = i ;
  return j ; // tra ve node cuoi trong pathnode
  }
```

Phép toán split :

Tách nút đầy pathnode [nd] thành hai nút : nút nửa trái pathnode [nd] và nút nửa phải nd2. Phép toán này cũng cập nhật nút nửa trái và tạo mới nút nửa phải.

```
  void split (int nd , int newkey, int newnode, int pos,int
*pnd2, int *pmidkey)
  {
```

```
    // truong hop chen newky va newnode vao node nua phai
     if (pos > Ndiv2)
     {
     copy (nd, Ndiv2+1, ORDER-2, nd+1 ; /* dung
     pathnode [nd+1] luu node nua phai */
     insnode (nd+1, newkey, newnode, pos-Ndiv2-1) ;
     // so nhanh cay con con lai cua node nua trai
     pathnode[nd].numtrees = Ndiv2+1 ;
     *pmidkey = pathnode [nd].key [Ndiv2] ;
     }
     // truong hop newkey la midkey
     if (pos == Ndiv2)
     {
     copy (nd, Ndiv2, ORDER-2, nd+1) ; /* dung
     pathnode [nd+1] luu node nua phai */
     // so nhanh cay con con lai cua node nua trai
     pathnode [nd]. Numtrees = Ndiv2+1 ;
     /* Dieu chinh lai node con dau tien cua node      nua phai
*/
     pathnode [nd+1].Branch [0] = newnode ;
     *pmidkey = newkey ;
     }
     /* truong hop chen newkey va newnode vao node nua trai */
     if (pos <Ndiv2)
     {
     copy (nd, Ndiv2, ORDER-2, nd+1) ;  /* dung
     pathnode [nd+1] luu node nua phai */
     //so nhanh cay con con lai cua node nua trai
     pathnode[nd].numtrees = Ndiv2 ;
     *pmidkey = pathnode[nd].key (Ndiv2-1) ;
     insnode (nd, newkey, newnode, pos) ;
     }
     /* cap nhat lai node nua trai pathnode[nd] va tao node nua
phai vao B-Tree */
     replace(nd,location[nd]) ;
     *pnd2 = makenode (nd+1);
    }
```

Phép toán insert :

Thêm khóa k vào vị trí position của nút lá pathnode[s]

```
    void insert (int s, int k, int position)
    {
        struct node x ;
        int nd, nd2, newkey, newnode, pos, midkey, i ;
        /* Khoi dong cac tri truoc khi vao vong lap tach cac
    node day pathnode[nd] */
        nd = s ;
        newkey = k ;
        newnode = -1 ;
        pos = position ;
        // Vong lap tach cac node day pathnode [nd]
        while (nd ! =0 && pathnode[nd].numtrees == ORDER)
```

```
        {
        split(nd,newkey, newnode, pos, &nd2, &midkey) ;
        // Gan lai cac tri sau lan tach node truoc
        newkey = midkey ;
        newnode = nd2 ;
        pos = nodesearch (nd-1, midkey) ;
    nd--; // len node cha
        }
        /* Truong hop node pathnode[nd] chua day va pathnode
[nd] khong phai la node goc */
        if (pathnode[nd].numtrees <ORDER)
        {
             /* chen newkey va newnode tai vi tri pos cua
node pathnode [nd] */
             insnode (nd,newkey, newnode, pos) ;
             return ;
        }
        /* Truong hop node pathnode[nd] la node goc bi
 day, tach node goc nay va tao node goc moi */
        split (nd, newkey, newnode, pos, &nd2, &midkey) ;
        x.numtrees = 2 ;
        x.key [0] = mikey ;
    // khoi dong dia chi cac node con cua node goc moi
    for (i = 0 < i < ORDER; i++)
    x.Branch[i] = -1 ;
    // Gan lai hai nhanh cay con cua node goc moi
    x. Branch[0] = location [nd] ;
    x. Branch[1] = nd2 ;
    Root = makeroot (x) ;
}
```

Chương trình minh họa :

Chương trình sau đây tổ chức B-Tree như một tập tin chứa các mẫu tin (struct), mỗi mẫu tin là một nút của B-Tree.

- Mẫu tin đầu tin được dành riêng để chứa header, header dùng lưu các thông tin quản lý B-Tree như Root-địa chỉ của nút gốc numnodes-số nút có trên B-Tree, depth-chiều sâu của B-Tree,…

- Các mẫu tin sau : chức các nút của B-Tree, mỗi nút có các trường sau :

numtrees ; // *số nhánh cây con của nút*

key[ORDER-1]; // *các khóa của nút*

Branch[ORDER] ;*// các địa chỉ của các nút con của nút*

Tham khảo chương trình minh họa B-Tree tổ chức ở bộ nhớ ngoài sau đây :

· **Chương trình minh họa B-Tree tổ chức ở bộ nhớ ngoài**

```
#include   <stdio.h>
#include <stdlib.h>
#include   <conio.h>
#include   <alloc.h>
#define ORDER 5 // bac cua B-Tree
#define Ndiv2 ORDER/2
#define TRUE 1
#define FALSE 0
// Khai bao cau truc mot node cua B-Tree o bo nho ngoai
```

```
struct node
    typedef struct node
    {
     int numtrees ; // so nhanh cay con cua node
     int key [ORDER-1] ; // cac khoa cua node
     int Branch [ORDER] ;//cac dia chi den node con
    };
    struct node pathnode [20] ;// mang chua cac node tren duong
di tu node goc den node la
     int location[20] ;// mang chua dia chi cac node tren
pathnode
     int Root ;          // dia chi node goc cua B-Tree
     // Tac vu initialise :   khoi dong B-Tree
    void initialize ()
     {
     FILE *f ;
     struct node header ;
     header.numtrees = -1 ;
     header.key [0] = 0 ;
     f = fopen ("c :\\btree,dat", "wb") ;
    fwrite (&header, sizeof (struct node), 1, f) ;
    fclose (f) ;
    }
    /* Tac vu readheader : doc cac thong tin trong header cua
tap tin B-Tree */
    void readheader ()
     {
     FILE *f ;
     struct node header ;
     f = fopen ("c:\\btree.dat", "rb") ;
     fread (&header, sizeof (struct node), 1, f) ;
     Root = header.numtrees ; /* truy xuat dia chi
    node goc cua B-Tree */
     fclose (f) ;
     }
    // Tac vu makeroot : tao node goc moi x cho B-Tree
    int makeroot (struct node x)
    {
    FILE *f ;
    struct node temp ;
    if ((f = fopen ("c:\\btree.dat", "r+b"))== NULL
     {
     printf("khong mo tap tin chua B-Tree duoc\n") ;
     return (0) ;
     }
    // cap nhat thong tin trong header
    fread (&temp, sizeof (struct node), 1, f) ;
    temp.numtrees = temp.key [0];//cap nhat lai dia chi node
goc moi
    temp.key [0] ++ ;     // tang so node co trong B-Tree
    fseek (f, 0, SEEK_SET) :
```

```
    ferite (&temp, sizeof (struct node),1,f) ;//ghi lai header
    // append node goc moi vao cuoi tap tin
    fseek (f, 0, SEEK_END) ;
    fwrite (&x, sizeof (struct node), 1, f) ;
    fclose (f) ;
    return (temp.key [0]-1);//tra ve dia chi cua node goc moi
    }
    /* Tac vu makenode: tao node moi pathnode [i] cho B-Tree
Node moi nay la node nua phai cua lan tach node */
    int makenode (int i)
     {
     FILE *f ;
     struct node temp ;
     if (( = fopen ("c:\\btree.dat", "r+b")) == NULL)
          {
          printf ("khong mo tap tin chua B-Tree duoc\n");
          return (0) ;
          }
     // cap nhat thong tin trong header
     fread (&temp, sizeof (struct node), 1, f) ;
     temp.key [0]++;// tang so node co trong B-Tree
     fseek (f, 0, SEEK_SET) ;
     fwrite (&temp, sizeof (struct node), 1, f) ; // ghi lai
header append node moi vao cuoi tap     tin
     fseek (f, 0, SEEK_END) ;
     fclose (f) ;
    return (temp.key[0]-1) ; // tra ve dia chi cua node moi
     }
    /* Tac vu access : Nap node tai dia chi loc vao pathnode
[i] , nap dia chi loc vao location [i] */
    void access (int i, int loc)
    {
    FILE *f ;
     if ((f = fopen ("c :\\btree.dat", "rb"))==NULL)
      {
          printf ("Khong mo tap tin chua B-Tree duoc\n") ;
          return ;
      }
    fseek (f, (loc+1) *sizeof (struct node), SEEK_SET) ;
    // Chep node tai dia chi loc vao pathnode [i]
    fread (&pathnode [i], sizeof (struct node), 1, f) ;
    // Chep dia chi loc vao location [i]
    location [i] = loc ;
    fclose (f) ;
     }
    /* Tac vu replace : cap nhat lai node pathnode[i] tai dia
chi loc cua tap tin B-Tree */
    void replace (int i, int loc)
    {
    FILE *f
    if ((f = fopen ("c:\\btree.dat", "r+b")) == NULL
```

```
    {
    prinf ("khong mo tap tin chua B-Tree duoc\n\n") ;
    return ;
    }
    fseek (f, (loc+1) * sizeof (struct node), SEEK_SET) ;
    fwrite (&pathnode [i], sizeof (struct node), 1, f) ;
    fclose (f) ;
    }
    /* Tac vu viewnodes : Xem chi tiet noi dung tung node trong
tap tin B-Tree */
    void viewnodes ()
    {
    FILE *f ;
    struct node x ;
    int i =0, j, numnodes ;
    f = fopen ("c: \\btree.dat", "rb") ;
    // Doc va hien thi header
    fread (&x, sizeof (struct node), 1 , f) ;
    numnodes = x.key [0] ;
    printf ("Dia chi node goc : %d so node tren B-Tree : %d",
x.numtrees, x.key             [0]);
    /* Doc va hien thi thong tin cua tung node trong B-Tree */
    while (i < numnodes)
      {
      printf ("\nDia chi %d :", i) ;
      fread (&x, sizeof (struct node), 1, f) ;
      printf (" bac : %d", x.numtrees) ;
      printf (" khoa va node con : ") ;
      for (j = 0 ; j < x.numtrees-1 ; j++)
      printf ("%d", x.key [j]) ;
      printf (" ") ;
      for (j = 0 ; j < x.numtrees ; j++)
      printf ("%d", x.Branch [j]) ;
      i++
      }
    fclose (f) ;
    }
    /*Tac vu nodesearch: tim vi tri trong node pathnode [p]
(tra ve vi tri cua khoa bat dau lon hon hay bang k).Truong hop
k lon hon tat ca cac khoa thi tra ve
    pathnode [p]. numtrees-1 */
    int nodesearch (int p, int k)
    {
    int i ;
    for (i =0 ; i <pathnode [p]. numtrees-1 &&
    pathnode [p].key [i] <k ; i++) ;
    return (i) ;
    }
    /* Tac vu search : tim khoa k tren B-Tree
    Neu co :
      Bien found tra ve gia tri TRUE
```

```
     Ham search () tra ve node vi tri j trong
     pathnode co chua khoa k
     Bien position tra ve vi tri cua khoa k co tren    node nay
   Neu khong co :
     Bien found tra ve gia tri FALSE
     Ham search () tra ve node vi tri cuoi trong
     pathnode la node co them khoa k vao
     Bien position tra ve vi tri cua khoa k
     neu them no vao node nay */
   int search (int k, int *pposition, int *pfound)
     {
     p = Root; // p xuat phat tu node goc cua B-Tree
     j = -1 ;// j la chi so tren pathnode
          while (p ! =-1)
          {
          j++ ;
          access (j, p) ;
          i = nodesearch (j, k) ;
   if (i < pathnode [j]). numtrees-1 && k == pathnode [j] .
key [i])
           {
               * pfound = TRUE ;
               * pposition = i
               return j ;
           }
   p = pathnode [j].Branch [i] ;/* p chuyen xuong node con */
     }
   /* truong hop khong co khoa k tren B-Tree,luc nay da hinh
thanh mang pathnode [] chua ca node tren duong di tu node goc
den node la */
     * pfound = FALSE ;
     * pposition = i ;
     return j ;     // tra ve node cuoi trong pathnode
   }
   /* Tac vu insnode : chen khoa newkey vao vi tri pos cua
node chua day pathnode [p], va chen nhanh cay con newnode vao
vi tri ben phai cua khoa newkey */
   void insnode(int p, int newkey,int newnode, int pos )
   {
   int i ;
   /* doi cac nhanh cay con va cac khoa tu vi tri pos tro ve
sau xuong mot vi tri */
   for (i = pathnode [p]. numtrees-1 ; i>=pos+1 ; i--)
   {
      pathnode [p].Branch [i+1] = pathnode [p].Branch [i] ;
      pathnode [p] .key [i] = pathnode [p] .key [i-1] ;
   }
   // Gan khoa newkey vao vi tri pos
   pathnode [p].key [pos] = newkey ;
   /* Gan nhanh newnode la nhanh cay con ben phai cua khoa
newkey */
```

```
    pathnode [p].Branch [pos +1] = newnode ;
    // Tang so nhanh cay con cua node p len 1
    pathnode [p].numtrees +=1 ;
    }
    /* Tac vu copy : chep cac khoa (va nhanh cay con) tu vi tri
first den sang node pathnode [nd1] (node nua phai) */
    void copy (int nd, int first, int last, int nd1)
    {
    int i ;
    //copy cac khoa tu node pathnode [nd] qua pathnode [nd1]
    for (i =first ; i <= last ; i++)
    pathnode [nd1].key [i-first] =pathnode [nd].key [i] ;
    /* copy cac nhanh cay con tu node pathnode [nd] qua
pathnode [nd1] */
    for (i = first; i <= last+1 ; i++)
    pathnode[nd1].Branch[i-first]=pathnode[nd].Branch[i] ;
    /* so nhanh cay con cua node pathnode [nd1] */
    pathnode [nd1].numtrees,= last=first+2 ;
    }
    /* Tac vu split : tach node day pathnode [nd] thanh hai
node : node nua trai pathnode [nd] va node nua phai nd2 */
    void split (int nd, int *pmidkey)
    {
    // truong hop chen newkey va newnode vao node nua phai
    if (pos > Ndiv2)
    {
     copy (nd, Ndiv2+1, ORDER-2, nd+1) ;
    /* dung pathnode [nd+1] luu node nua phai */
     insnode (nd+1, newkey, newnode, pos-Ndiv2-1 ;
     // so nhanh cay con con lai cua node nua trai
     pathnode [nd].numtrees = Ndiv2+1 ;
      *pmidkey = pathnode [nd].key [Ndiv2] ;
    }
      // Truong hop newkey la midkey
     if (pos == Ndiv2)
     {
     copy (nd, Ndiv2, ORDER-2, nd+1) ;
     //dung pathnode [nd+1] luu node nua phai
     /* so nhanh cay con con lai cua node nua trai pathnode
[nd].numtrees = Ndiv2+1.Dieu chinh lai node con dau tien cua
node nua phai */
     pathnode [nd+1].Branch [0] = newnode ;
     * pmidkey = newkey ;
     }
     // truong hop chen newkey va newnode vao node nua trai
     if (pos <Ndiv2)
     {
     copy (nd, Ndiv2, ORDER-2, nd+1) ;  /*dung
     pathnode [nd+1] luu node nua phai */
     // so nhanh cay con con lai cua node nua trai
     pathnode [nd],numtrees = Ndiv2 ;
```

```
     * pmidkey = pathnode [nd].key [Ndiv2-1] ;
     insnode (nd, newkey, newnode, pos) ;
     }
     /* cap nhat lai node nua trai pathnode [nd] va tao node
nua phai vao tap tin B-Tree */
     replace (nd, location [nd]) :
     * pnd2 = makenode (nd+1) ;
    }

    /* Tac vu insert : Them khoa k vao vi tri position cua node
la pathnode [s] */
    void insert (int s, int k, int position)
    {
     struct node x ;
     int nd, nd2, newkey, newnode, pos, midkey, i ;
     /* Khoi dong cac tri truoc khi vao vong lap  tach cac node
day pathnode [nd] */
     nd = s ;
     newkey = k ;
    newnode = -1 ;
    pos = position ;
    // Vong lap tach cac node day pathnode [nd]
    while (nd !=0 && pathnode [nd].numtrees == ORDER)
     {
     split (nd,newkey,newnode, pos, &nd2, &midkey) ;
     // Gan lai cac tri sau lan tach node truoc
     newkey = midkey ;
     newnode = nd2 ;
     pos = nodesearch (nd-1, midkey) ;
     nd-- ;
     }
    /* Truong hop node pathnode [nd] chua day va pathnode [nd]
khong phai la node goc */
    if (pathnode [nd].numtrees <ORDER)
     {
    /* chen newkey va newnode tai vi tri phos cua node pathnode
[nd] */
    insnode (nd, newkey, newnode, pos) ;
    replace (nd, location [nd]) ;
    return ;
     }
    /* Truong hop node pathnode [nd] la node goc bi day, tach
node goc nay va tao node goc moi */
    split (nd, newkey, newnode, pso, &nd2, &midkey) ;
    x.numtrees = 2 ;
    x. key [0] = midkey ;
    // khoi dong dia chi cac node con cua node goc moi
    for (i = 0 ; i <ORDER ; i++)
    x.Branch [i] = -1 ;
    // Gan lai hai nhanh cay con cua node goc moi
    x.Branch [0] = location [nd] ;
```

```
    x.Branch [1] = nd2 ;
    Root = makeroot (x) ;
    }

    // Chuong trinh chinh
    void main ()
    {
    struct node x ;
    int s, i, n, k, pos, timthay, chucnang ;
    char c ;
    clrscr () ;
    //  Doc  cac  thong  tin  trong  header  cua  tap  tin  B-
Treereadheader () ;
    do
    {
    printf ("\n\nCHUONG TRINH MINH HOA B-TREE TO CHUC O BO NHO
NGOAI");
    printf ( "\n\nCac chuc nang cua chuong trinh : \n") ;
    printf ("   1 : Them mot khoa\n") ;
    printf ("   2 : Them ngau nhien nhieu khoa\n") ;
    printf ("   3 : Xem tap tin B-Tree\n") ;
    printf ("   4 : Khoi dong B-Tree \n") ;
    printf ("   5 : Tim kiem\n") ;
    printf ("   0 : Ket thuc chuong trinh\n") ;
    printf ("Chuc nang ban chon :") ;
    scanf ("%d", &chucnang) :
    switch (chucnang)
    {
    case 1 :
    {
    printf ("\nNoi dung khoa moi :") ;
       scanf (" %d", &k) ;
      // Truong hop B-Tree bi rong thi tao node goc
       if (Root == -1)
           {
                x.numtrees = 2
                x. key [0] = k ;
                for (i = 0 ; i < ORDER; i++)
                x.Branch [i] = -1 ;
                Root = makeroot (x) ;
           }
       else
           {
                s = search (k, &pos, &timthay) ;
                if (timthay == TRUE)
                 printf ("TRung khoa, khong them khoa d vao B-
Tree duoc",k) ;
                else
    insert (s, k, pos);
    /* them khoa k vao node la pathnode [s] tai vi tri pos */
    }
```

```
      break ;
    }
    case 2 :
    {
      printf ("\n So khoa them vao?");
      scanf ("% d", &n) ;
      for (i = 0 ; i < n ; i++)
      {
      k = random (10000) ;
      s = search (k, &pos, &timthay) ;
      if (timthay == TRUE)
    printf ("\nBi trung khoa, khong them khoa %d vao B-Tree
duoc", k);
    else
      insert (s, k, pos) ;
      }
      printf ("\n Da them vao B-Tree &d so ngau nhien", n);
      break ;
    }
    case 3 :
    {
      printf ("\nXem tap tin B-Tree\n") ;
      viewnodes () ;
      break ;
    }
    case 4 :
    {
    initialize () ;
      Root = -1 ;
      break ;
    }
    case 5 :
    {
      printf ("\nkhoa can tim : ") ;
      scanf (" %d", &k) ;
      s = search (k, &pos, &timthay) ;
    if (timthay == TRUE)printf ("khong tim thay") ;
      break ;
    }
    }
    } while (chucnang ! = 0) ;
    }
```

Đánh giá hiệu quả B-tree

Bởi vì có nhiều mẫu tin trên một nút, và nhiều node trên cùng một mức, do đó các thao tác trên B-tree phải rất nhanh vì dữ liệu được lưu trữ trên đĩa. Trong ví dụ danh bạ điện thoại có 500,000 mẫu tin. Tất cả các node trong B-tree đều có ít nhất một nửa là node đầy, vì thế chúng chứa đựng ít nhất 8 mẫu tin và 9 liên kết đến các node con. Kết quả là chiều cao của cây khoảng nhỏ hơn $\log_9 N$ (logarit cơ số 9 của N), trong đó N là 500,000. Logarit này sẽ có giá trị là 5.972, vì thế cây sẽ có 6 mức.

Kết quả là, khi sử dụng B-tree chỉ cần 6 lần truy cập đĩa để tìm bất kì mẫu tin nào trong tập tin chứa 500,000 mẫu tin. Với 10 milli giây trên một lần truy cập, sẽ chiếm khoảng 60

milli giây hay 6/100 giây. Kết quả này nhanh hơn rất nhiều so với việc tìm kiếm nhị phân trong tập tin có thứ tự tuần tự.

Càng nhiều mẫu tin trong một nút, cây sẽ có càng ít các mức. Như chúng ta đã thấy ở ví dụ trên chỉ cần 6 mức cho 500,000 mẫu tin, dù rằng mỗi node chỉ lưu trữ có 16 mẫu tin. Ngược lại, trong cây nhị phân với 500,000 mục dữ liệu sẽ cần khoảng 19 mức. Nếu chúng ta sử dụng các khối với hàng trăm mẫu tin, chúng ta có thể giảm số lượng các mức trong cây xuống và cải tiến được số truy cập đĩa.

Không những việc tìm kiếm trong B-tree nhanh hơn so với trong tập tin có thứ tự trên đĩa, mà đối với việc chèn và xóa ở B-tree cũng hiệu quả hơn rất nhiều.

Xét việc chèn vào trong B-tree mà không cần phải tách nút. Đây là tình huống thông dụng nhất, bởi vì một số lượng lớn các mẫu tin nằm trên một nút. Trong ví dụ danh bạ điện thoại (như chúng ta đã thấy) chỉ cần 6 lần truy cập để tìm kiếm điểm chèn. Khi đó sẽ có nhiều hơn một lần truy cập để ghi khối mà chứa đựng mẫu tin mới cần chèn vào đĩa; Tổng cộng là có 7 lần truy cập.

Chúng ta sẽ xem xét việc tách nút. Node bị tách phải được đọc (có một nửa các mẫu tin của nó bị xoá đi) và ghi lại đĩa. Node mới tạo ra phải được ghi vào đĩa, và node cha của nó phải được đọc (tương tự như việc chèn mẫu tin nâng lên) và ghi lại đĩa. Điều này cần thêm 5 lần truy cập cộng với 6 lần truy cập để tìm kiếm điểm chèn, tổng cộng ta có 12 lần. Đây là sự cải tiến đáng kể so với 500,000 lần truy cập cho việc chèn vào tập tin tuần tự.

Trong các B-tree cải tiến, chỉ có các node lá mới chứa các mẫu tin. Các node không phải là lá chỉ cần chứa các khoá và các số của khối. Điều này có thể thực hiện thao tác nhanh hơn bởi vì mỗi khối có thể lưu nhiều hơn các số của khối. Kết quả là cây có bậc cao hơn sẽ có ít mức hơn, và tốc độ truy cập sẽ được tăng lên. Tuy nhiên, việc lập trình sẽ trở nên phức tạp bởi vì có 2 loại nút: node lá và không phải node lá.

❖ Bài tập củng cố:

1. Cho dãy số:

27 9 17 35 16 32 11 22 25 6 33 1 19 3 12 28 14 8 20 30 23 4

a. Xây dựng B cây bậc 5 cho dãy số trên (vẽ cây).

b. Vẽ lại B cây ở câu a. sau khi xóa lần lượt các khóa 3 và 35.

c. Vẽ lại B cây ở câu a. sau khi thêm lần lượt các khóa 40 và 10.

2. Cài đặt các phép toán cho B-Cây.

CHƯƠNG 5
KỸ THUẬT THIẾT KẾ GIẢI THUẬT

❖ **Mục tiêu học tập:** Sau khi học xong chương này, người học có thể:
- Nắm vững nội dung các kỹ thuật thiết kế giải thuật: chia để trị, tham ăn, quy hoạch động, quay lui và tìm kiếm địa phương.
- Vận dụng các kỹ thuật vào giải các bài toán thực tế.
- Đánh giá được giải thuật trên các kỹ thuật thiết kế.

Khi thiết kế một giải thuật chúng ta thường dựa vào một số kỹ thuật nào đó. Chương này sẽ trình bày một số kỹ thuật quan trọng để thiết kế giải thuật như: Chia để trị (Divide-and-Conquer), quy hoạch động (dynamic programming), kỹ thuật tham ăn (greedy techniques), quay lui (backtracking) và tìm kiếm địa phương (local search). Các kỹ thuật này được áp dụng vào một lớp rộng các bài toán, trong đó có những bài toán cổ điển nổi tiếng như bài toán tìm đường đi ngắn nhất của người giao hàng, bài toán cây phủ tối tiểu...

5.1. KỸ THUẬT CHIA ĐỂ TRỊ

5.1.1 Nội dung kỹ thuật

Có thể nói rằng kỹ thuật quan trọng nhất, được áp dụng rộng rãi nhất để thiết kế các giải thuật có hiệu quả là kỹ thuật "chia để trị" (divide and conquer). Nội dung của nó là: Để giải một bài toán kích thước n, ta chia bài toán đã cho thành một số bài toán con có kích thước nhỏ hơn. Giải các bài toán con này rồi tổng hợp kết quả lại để được lời giải của bài toán ban đầu. Đối với các bài toán con, chúng ta lại sử dụng kỹ thuật chia để trị để có được các bài toán kích thước nhỏ hơn nữa. Quá trình trên sẽ dẫn đến những bài toán mà lời giải chúng là hiển nhiên hoặc dễ dàng thực hiện, ta gọi các bài toán này là **bài toán cơ sở.**

Tóm lại kỹ thuật chia để trị bao gồm hai quá trình: Phân tích bài toán đã cho thành các bài toán cơ sở và tổng hợp kết quả từ bài toán cơ sở để có lời giải của bài toán ban đầu. Tuy nhiên đối với một số bài toán, thì quá trình phân tích đã chứa đựng việc tổng hợp kết quả do đó nếu chúng ta đã giải xong các bài toán cơ sở thì bài toán ban đầu cũng đã được giải quyết. Ngược lại có những bài toán mà quá trình phân tích thì đơn giản nhưng việc tổng hợp kết quả lại rất khó khăn. Trong các phần tiếp sau ta sẽ trình bày một số ví dụ để thấy rõ hơn điều này.

Kỹ thuật này sẽ cho chúng ta một giải thuật đệ quy mà việc xác định độ phức tạp của nó sẽ phải giải một phương trình đệ quy như trong chương I đã trình bày.

5.1.2 Nhìn nhận lại giải thuật MergeSort và QuickSort

Hai giải thuật sắp xếp đã được trình bày trong cấu trúc dữ liệu và giải thuật 1 (MergeSort và QuickSort) thực chất là đã sử dụng kỹ thuật chia để trị.Với MergeSort, để sắp một danh sách L gồm n phần tử, chúng ta chia L thành hai danh sách con L1 và L2 mỗi danh sách có n/2 phần tử. Sắp xếp L1, L2 và trộn hai danh sách đã được sắp này để được một danh sách có thứ tự. Quá trình phân tích ở đây là quá trình chia đôi một danh sách, quá trình này sẽ dẫn đến bài toán sắp xếp một danh sách có độ dai bằng 1, đây chính là bài toán cơ sở vì việc sắp xếp danh sách này là "không làm gì cả". Việc tổng hợp các kết quả ở đây là "trộn 2 danh sách đã được sắp để được một danh sách có thứ tự".

Với QuickSort, để sắp xếp một danh sách gồm n phần tử, ta tìm một giá trị chốt và phân hoạch danh sách đã cho thành hai danh sách con "bên trái" và "bên phải ". Sắp xếp "bên trái"

và "bên phải" thì ta được danh sách có thứ tự. Quá trình phân chia sẽ dẫn đến các bài toán sắp xếp một danh sách chỉ gồm một phần tử hoặc gồm nhiều phần tử có khoá bằng nhau, đó chính là các bài toán cơ sở, vì bản thân chúng đã có thứ tự rồi. Ở đây chúng ta cũng không có việc tổng hợp kết quả một cách tường minh, vì việc đó đã được thực hiện trong quá trình phân hoạch.

5.1.3 Bài toán nhân các số nguyên lớn

Trong các ngôn ngữ lập trình đều có kiểu dữ liệu số nguyên (chẳng hạn kiểu integer trong Pascal, Int trong C…), nhưng nhìn chung các kiểu này đều có miền giá trị hạn chế (chẳng hạn từ -32768 đến 32767) nên khi có một ứng dụng trên số nguyên lớn (hàng chục, hàng trăm chữ số) thì kiểu số nguyên định sẵn không đáp ứng được. Trong trường hợp đó, người lập trình phải tìm một cấu trúc dữ liệu thích hợp để biểu diễn cho một số nguyên, chẳng hạn ta có thể dùng một chuỗi kí tự để biểu diễn cho một số nguyên, trong đó mỗi kí tự lưu trữ một chữ số. Để thao tác được trên các số nguyên được biểu diễn bởi một cấu trúc mới, người lập trình phải xây dựng các phép toán cho số nguyên như phép cộng, phép trừ, phép nhân… Sau đây ta sẽ đề cập đến bài toán nhân hai số nguyên lớn.

Xét bài toán nhân hai số nguyên lớn X và Y, mỗi số có n chữ số.

Đầu tiên ta nghĩ đến giải thuật nhân hai số thông thường, nghĩa là nhân từng chữ số của X với số Y rồi cộng các kết quả lại. Việc nhân từng chữ số của X với sô Y đòi hỏi phải nhân từng chữ số của X với từng chữ số của Y, vì X và Y đều có n chữ số nên cần n^2 phép nhân hai chữ số, mỗi phép nhân hai chữ số này tốn O(1) thì phép nhân cũng tốn $O(n^2)$ thời gian.

Áp dụng kỹ thuật "chia để trị" vào phép nhân các số nguyên lớn, ta chia mỗi số nguyên lớn X và Y thành các số nguyên lớn có n/2 chữ số. Để đơn giản cho việc phân tích giải thuật ta giả sử **n là luỹ thừa của 2**, còn về khía cạnh lập trình, ta vẫn có thể viết chương trình với n bất kì.

$X = A10^{n/2} + B$ và $Y = C10^{n/2} + D$

Trong đó A, B, C, D là các số nguyên lớn có n/2 chữ số.

Chẳng hạn với X = 1234 thì A = 12 và B = 34 bởi vì $X = 12 * 10^2 + 34$.

Khi đó tích của X và Y là: $XY = AC10^n + (AD + BC)10^{n/2} + BD$ (1) Với mỗi số có n/2 chữ số, chúng ta lại tiếp tục phân tích theo cách trên, quá trình phân tích sẽ dẫn đến bài toán cơ sở là nhân các số nguyên lớn chỉ gồm một chữ số mà ta dễ dàng thực hiện. Việc tổng hợp kết quả chính là thực hiện các phép toán theo công thức (1).

Theo (1) thì chúng ta phải thực hiện 4 phép nhân các số nguyên lớn n/2 chữ số (AC, AD, BC, BD), sau đó tổng hợp kết quả bằng 3 phép cộng các số nguyên lớn n chữ số và 2 phép nhân với 10^n và $10^{n/2}$.

Các phép cộng các số nguyên lớn n chữ số dĩ nhiên chỉ cần O(n). Phép nhân với 10^n có thể thực hiện một cách đơn giản bằng cách thêm vào n chữ số 0 và do đó cũng chỉ lấy O(n). Gọi T(n) là thời gian để nhân hai số nguyên lớn, mỗi số có n chữ số thì từ (1) ta có phương trình đệ quy:

$T(1) = 1$

$T(n) = 4T(n/2) + cn$ (2)

Giải (2) ta được $T(n) = O(n^2)$. Như vậy thì chẳng cải tiến được chút nào so với giải thuật nhân hai số bình thường. Để cải thiện tình hình, chúng ta có thể viết lại (1) thành dạng:

$XY = AC10^n + [(A-B)(D-C) + AC + BD] 10^{n/2} + BD$ (3)

Công thức (3) chỉ đòi hỏi 3 phép nhân của các số nguyên lớn n/2 chữ số là: AC, BD và (A-B)(D-C), 6 phép cộng trừ và 2 phép nhân với 10^n. Các phép toán này đều lấy O(n) thời gian. Từ (3) ta có phương trình đệ quy:

$T(1) = 1$

$T(n) = 3T(n/2) + cn$

Giải phương trình đệ quy này ta được nghiệm $T(n) = O(n^{\log 3}) = O(n^{1.59})$. Giải thuật này

rõ ràng đã được cải thiện rất nhiều.

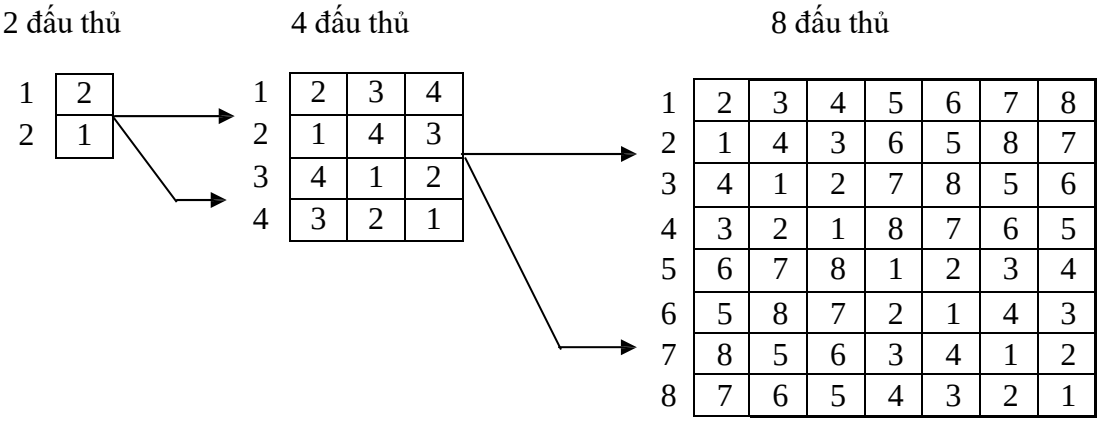
5.1.4. Xếp lịch thi đấu thể thao

Kỹ thuật chia để trị không những chỉ có ứng dụng trong thiết kế giải thuật mà còn trong nhiều lĩnh vực khác của cuộc sống. Chẳng hạn xét việc xếp lịch thi đấu thể thao theo thể thức đấu vòng tròn 1 lượt cho n đấu thủ. Mỗi đấu thủ phải đấu với các đấu thủ khác, và mỗi đấu thủ chỉ đấu nhiều nhất một trận mỗi ngày. Yêu cầu là xếp một lịch thi đấu sao cho số ngày thi đấu là ít nhất. Ta dễ dàng thấy rằng tổng số trận đấu của toàn giải là n(n - 1)/2. Như vậy nếu n là một số chẵn thì ta có thể sắp n/2 cặp thi đấu trong một ngày và do đó cần ít nhất n-1 ngày. Ngược lại nếu n là một số lẻ thì n-1 là một số chẵn nên ta có thể sắp (n-1)/2 cặp thi đấu trong một ngày và do đó ta cần n ngày. Giả sử $n = 2^k$ thì n là một số chẵn và do đó cần tối thiểu n-1 ngày.

Lịch thi đấu là một bảng n dòng và n-1 cột. Các dòng được đánh số từ 1 đến n và các cột được đánh số từ 1 đến n-1, trong đó dòng i biểu diễn cho đấu thủ i, cột j biểu diễn cho ngày thi đấu j và ô(i,j) ghi đấu thủ phải thi đấu với đấu thủ i trong ngày j. Chiến lược chia để trị xây dựng lịch thi đấu như sau: Để sắp lịch cho n đấu thủ, ta sẽ sắp lịch cho n/2 đấu thủ, để sắp lịch cho n/2 đấu thủ, ta sẽ sắp lịch cho n/4 đấu thủ... Quá trình này sẽ dẫn đến bài toán cơ sở là sắp lịch thi đấu cho 2 đấu thủ. Hai đấu thủ này sẽ thi đấu một trận trong một ngày, lịch thi đấu cho họ thật dễ sắp. Khó khăn chính là ở chỗ từ các lịch thi đấu cho hai đấu thủ, ta tổng hợp lại để được lịch thi đấu của 4 đấu thủ, 8 cấu thủ, ...

Xuất phát từ lịch thi đấu cho hai đấu thủ ta có thể xây dựng lịch thi đấu cho 4 đấu thủ như sau: Lịch thi đấu cho 4 đấu thủ sẽ là một bảng 4 dòng, 3 cột. Lịch thi đấu cho 2 đấu thủ 1 và 2 trong ngày thứ 1 chính là lịch thi đấu của hai đấu thủ (bài toán cơ sở). Như vậy ta có ô(1,1) = "2" và ô(2,1) = "1". Tương tự ta có lịch thi đấu cho 2 đấu thủ 3 và 4 trong ngày thứ 1. Nghĩa là ô(3,1) ="4" và ô(4,1) = "3". (Ta có thể thấy rằng ô(3,1) = ô(1,1) + 2 và ô(4,1) = ô(2,1) + 2). Bây giờ để hoàn thành lịch thi đấu cho 4 đấu thủ, ta lấy góc trên bên trái của bảng lắp vào cho góc dưới bên phải và lấy góc dưới bên trái lắp cho góc trên bên phải.

Lịch thi đấu cho 8 đấu thủ là một bảng gồm 8 dòng, 7 cột. Góc trên bên trái chính là lịch thi đấu trong 3 ngày đầu của 4 đấu thủ từ 1 đến 4. Các ô của góc dưới bên trái sẽ bằng các ô tương ứng của góc trên bên trái cộng với 4. Đây chính là lịch thi đấu cho 4 đấu thủ 5, 6, 7 và 8 trong 3 ngày đầu. Bây giờ chúng ta hoàn thành việc sắp lịch bằng cách lấp đầy góc dưới bên phải bởi góc trên bên trái và góc trên bên phải bởi góc dưới bên trái.



1	2	
2	1	

1	2	3	4
2	1	4	3
3	4	1	2
4	3	2	1

1	2	3	4	5	6	7	8
2	1	4	3	6	5	8	7
3	4	1	2	7	8	5	6
4	3	2	1	8	7	6	5
5	6	7	8	1	2	3	4
6	5	8	7	2	1	4	3
7	8	5	6	3	4	1	2
8	7	6	5	4	3	2	1

Hình 5.1: Lịch thi đấu của 2, 4 và 8 đấu thủ

5.1.5. Bài toán con cân bằng (Balancing Subproblems)

Đối với kỹ thuật chia để trị, nói chung sẽ tốt hơn nếu ta chia bài toán cần giải thành các bài toán con có kích thước gần bằng nhau. Ví dụ, sắp xếp trộn (MergeSort) phân chia bài toán thành hai bài toán con có cùng kích thước n/2 và do đó thời gian của nó chỉ là O(nlogn). Ngược lại trong trường hợp xấu nhất của QuickSort, khi mảng bị phân hoạch lệch thì thời

gian thực hiện là $O(n^2)$.

Nguyên tắc chung là chúng ta tìm cách chia bài toán thành các bài toán con có kích thước xấp xỉ bằng nhau thì hiệu suất sẽ cao hơn.

5.2. KỸ THUẬT "THAM ĂN"

5.2.1 Bài toán tối ưu tổ hợp

Là một dạng của bài toán tối ưu, nó có dạng tổng quát như sau:

- Cho hàm f(X) = xác định trên một tập hữu hạn các phần tử D. Hàm f(X) được gọi là hàm mục tiêu.
- Mỗi phần tử X ∈ D có dạng X = (x1, x2, .. xn) được gọi là một phương án.
- Cần tìm một phương án X ∈ D sao cho hàm f(X) đạt min (max). Phương án X như thế được gọi là phương án tối ưu.

Ta có thể tìm thấy phương án tối ưu bằng phương pháp "vét cạn" nghĩa là xét tất cả các phương án trong tập D (hữu hạn) để xác đinh phương án tốt nhất. Mặc dù tập hợp D là hữu hạn nhưng để tìm phương án tối ưu cho một bài toán kích thước n bằng phương pháp "vét cạn" ta có thể cần một thời gian mũ.

Các phần tiếp theo của chương này sẽ trình bày một số kỹ thuật giải bài toán tối ưu tổ hợp mà thời gian có thể chấp nhận được.

5.2.2 Nội dung kỹ thuật tham ăn

Tham ăn hiểu một cách dân gian là: trong một mâm có nhiều món ăn, món nào ngon nhất ta sẽ ăn trước và ăn cho hết món đó thì chuyển sang món ngon thứ hai, lại ăn hết món ngon thứ hai này và chuyển sang món ngon thứ ba…

Kỹ thuật tham ăn thường được vận dụng để giải bài toán tối ưu tổ hợp bằng cách xây dựng một phương án X. Phương án X được xây dựng bằng cách lựa chọn từng thành phần Xi của X cho đến khi hoàn chỉnh (đủ n thành phần). Với mỗi Xi, ta sẽ chọn Xi tối ưu. Với cách này thì có thể ở bước cuối cùng ta không còn gì để chọn mà phải chấp nhận một giá trị cuối cùng còn lại.

Áp dụng kỹ thuật tham ăn sẽ cho một giải thuật thời gian đa thức, tuy nhiên nói chung chúng ta chỉ đạt được **một phương án tốt chứ chưa hẳn là tối ưu**.

Có rất nhiều bài toán mà ta có thể giải bằng kỹ thuật này, sau đây là một số ví dụ.

5.2.3 Bài toán trả tiền của máy rút tiền tự động ATM.

Trong máy rút tiền tự động ATM, ngân hàng đã chuẩn bị sẵn các loại tiền có mệnh giá 100.000 đồng, 50.000 đồng, 20.000 đồng và 10.000 đồng. Giả sử mỗi loại tiền đều có số lượng không hạn chế. Khi có một khách hàng cần rút một số tiền n đồng (tính chẵn đến 10.000 đồng, tức là n chia hết cho 10000). Hãy tìm một phương án trả tiền sao cho trả đủ n đồng và số tờ giấy bạc phải trả là ít nhất.

Gọi X = (X1, X2, X3, X4) là một phương án trả tiền, trong đó X1 là số tờ giấy bạc mệnh giá 100.000 đồng, X2 là số tờ giấy bạc mệnh giá 50.000 đồng, X3 là số tờ giấy bạc mệnh giá 20.000 đồng và X4 là số tờ giấy bạc mệnh giá 10.000 đồng. Theo yêu cầu ta phải có X1 + X2 + X3 + X4 nhỏ nhất và X1*100.000 + X2*50.000 + X3*20.000 + X4*10.000 = n.

Áp dụng kỹ thuật tham ăn để giải bài toán này là: để có số tờ giấy bạc phải trả (X1 + X2 + X3 + X4) nhỏ nhất thì các tờ giấy bạc mệnh giá lớn phải được chọn nhiều nhất.

Trước hết ta chọn tối đa các tờ giấy bạc mệnh giá 100.000 đồng, nghĩa là X1 là số nguyên lớn nhất sao cho X1 * 100.000 ≤ n. Tức là X1 = n DIV 100.000.

Xác định số tiền cần rút còn lại là hiệu n – X1 * 100000 và chuyển sang chọn loại giấy bạc 50.000 đồng… Ví dụ khách hàng cần rút 1.290.000 đồng (n = 1290000), phương án trả tiền như sau:

X1 = 1290000 DIV 100000 = 12.

Số tiền cần rút còn lại là 1290000 – 12 * 100000 = 90000.
X2 = 90000 DIV 50000 = 1.
Số tiền cần rút còn lại là 90000 – 1 * 50000 = 40000.
X3 = 40000 DIV 20000 = 2.
Số tiền cần rút còn lại là 40000 – 2 * 20000 = 0.
X4 = 0 DIV 10000 = 0.

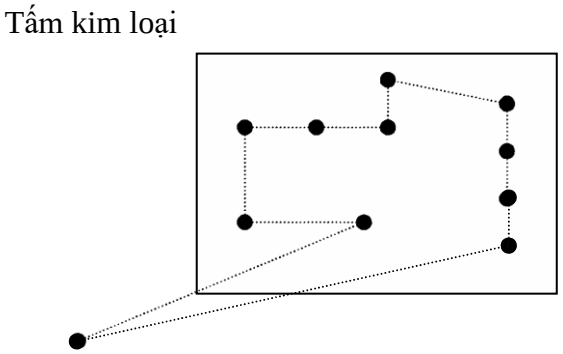
Ta có X = (12, 1, 2, 0), tức là máy ATM sẽ trả cho khách hàng 12 tờ 100.000 đồng, 1 tờ 50.000 đồng và 2 tờ 20.000 đồng.

5.2.4. Bài toán đường đi của người giao hàng

Chúng ta sẽ xét một bài toán rất nổi tiếng có tên là bài toán tìm đường đi của người giao hàng (TSP - Traveling Salesman Problem): Có một người giao hàng cần đi giao hàng tại n thành phố. Xuất phát từ một thành phố nào đó, đi qua các thành phố khác để giao hàng và trở về thành phố ban đầu. Mỗi thành phố chỉ đến một lần, khoảng cách từ một thành phố đến các thành phố khác là xác định được. Giả thiết rằng mỗi thành phố đều có đường đi đến các thành phố còn lại. Khoảng cách giữa hai thành phố có thể là khoảng cách địa lý, có thể là cước phí di chuyển hoặc thời gian di chuyển. Ta gọi chung là độ dài. Hãy tìm một chu trình (một đường đi khép kín thỏa mãn điều kiện trên) sao cho tổng độ dài các cạnh là nhỏ nhất. Hay còn nói là tìm một phương án có giá nhỏ nhất. Bài toán này cũng được gọi là bài toán người du lịch. Một cách tổng quát, có thể không tồn tại một đường đi giữa hai thành phố a và b nào đó. Trong trường hợp đó ta cho một đường đi ảo giữa a và b với độ dài bằng ∞.

Bài toán có thể biểu diễn bởi một đồ thị vô hướng có trọng số G = (V,E), trong đó mỗi thành phố được biểu diễn bởi một đỉnh, cạnh nối hai đỉnh biểu diễn cho đường đi giữa hai thành phố và trọng số của cạnh là khoảng cách giữa hai thành phố. Một chu trình đi qua tất cả các đỉnh của G, mỗi đỉnh một lần duy nhất, được gọi là chu trình Hamilton. Vấn đề là tìm một chu trình Hamilton mà tổng độ dài các cạnh là nhỏ nhất.

Bài toán này có những ứng dụng rất quan trọng. Thí dụ một máy hàn các điểm được điều khiển bởi máy tính. Nhiệm vụ của nó là hàn một số điểm dự định ở trên một tấm kim loại. Người thợ hàn bắt đầu từ một điểm bên ngoài tấm kim loại và kết thúc tại chính điểm này, do đó tấm kim loại phải được di chuyển để điểm cần hàn được đưa vào vị trí hàn (tương tự như ta đưa tấm vải vào đầu mũi kim của máy khâu). Cần phải tìm một phương án di chuyển tấm kim loại sao cho việc di chuyển ít nhất. Hình ảnh sau cho chúng ta hình dung về bài toán đặt ra.



Hình 5.2: Hàn các điểm trên một tấm kim loại

Dễ dàng thấy rằng, có thể áp dụng bài toán đường đi của người giao hàng để giải bài toán này.

Với phương pháp vét cạn ta xét tất cả các chu trình, mỗi chu trình tính tổng độ dài các cạnh của nó rồi chọn một chu trình có tổng độ dài nhỏ nhất. Tuy nhiên chúng ta cần xét tất cả

là (n - 1)! chu trình. Thực vậy, do mỗi chu trình đều đi qua tất cả các 2 đỉnh (thành phố) nên ta có thể cố định một đỉnh. Từ đỉnh này ta có n-1 cạnh tới n-1 đỉnh khác, nên ta có n-1 cách chọn cạnh đầu tiên của chu trình. Sau khi đã chọn được cạnh đầu tiên, chúng ta còn n-2 cách chọn cạnh thứ hai, do đó ta có (n-1)(n-2) cách chọn hai cạnh. Cứ lý luận như vậy ta sẽ thấy có (n-1)! cách chọn một chu trình. Tuy nhiên với mỗi chu trình ta chỉ quan tâm đến tổng độ dài các cạnh chứ không quan tâm đến hướïng đi theo chiều dương hay âm vì vậy có tất cả (n - 1)! phương án. Đó là một giải thuật thời gian mũ!.

Kỹ thuật tham ăn áp dụng vào đây là:

1. Sắp xếp các cạnh theo thứ tự tăng của độ dài.
2. Xét các cạnh có độ dài từ nhỏ đến lớn để đưa vào chu trình.
3. Một cạnh sẽ được đưa vào chu trình nếu cạnh đó thỏa mãn hai điều kiện sau:

• Không tạo thành một chu trình thiếu (không đi qua đủ n đỉnh)

• Không tạo thành một đỉnh có cấp ≥ 3 (tức là không được có nhiều hơn hai cạnh xuất phát từ một đỉnh, do yêu cầu của bài toán là mỗi thành phố chỉ được đến một lần: một lần đến và một lần đi)

4. Lặp lại bước 3 cho đến khi xây dựng được một chu trình.

Với kỹ thuật này ta chỉ cần n(n-1)/2 phép chọn nên ta có một giải thuật cần $O(n^2)$ thời gian.

Ví dụ 5.1: Cho bài toán TSP với 6 đỉnh được cho bởi các tọa độ như sau:

• c(1,7) • d(15,7)

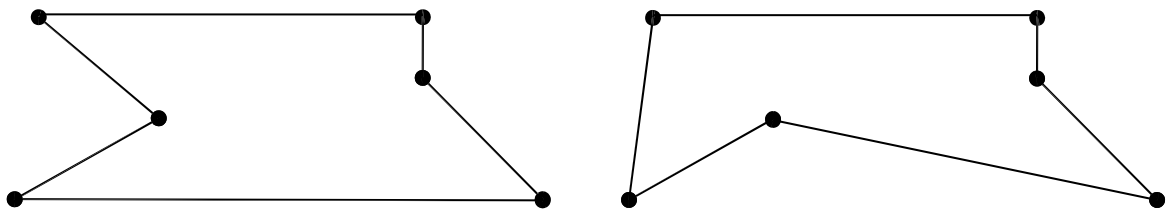
• b(4,3) • e(15,4)

• a(0,0) • f(18,0)

Hình 5.3: Sáu thành phố được cho bởi toạ độ

Do có 6 đỉnh nên có tất cả 15 cạnh. Đó là các cạnh: ab, ac, ad, ae, af, bc, bd, be, bf, cd, ce, cf, de, df và ef. Độ dài các cạnh ở đây là khoảng cách Euclide. Trong 15 cạnh này thì de = 3 là nhỏ nhất, nên de được chọn vào chu trình. Kế đến là 3 cạnh ab, bc và ef đều có độ dài là 5. Cả 3 cạnh đều thỏa mãn hai điều kiện nói trên, nên đều được chọn vào chu trình. Cạnh có độ dài nhỏ kế tiếp là ac = 7.08, nhưng không thể đưa cạnh này vào chu trình vì nó sẽ tạo ra chu trình thiếu (a-b-c-a). Cạnh df cũng bị loại vì lý do tương tự. Cạûnh be được xem xét nhưng rồi cũng bị loại do tạo ra đỉnh b và đỉnh e có cấp 3. Tương tự chúng ta cũng loại bd. cd là cạnh tiếp theo được xét và được chọn. Cuối cùng ta có chu trình a-b-c-d-e-f-a với tổng độ dài là 50. Đây chỉ là một phương án tốt.

Phương án tối ưu là chu trình a-c-d-e-f-b-a với tổng độ dài là 48.39.



Hình 5.4: Phương án Greedy và phương án tối ưu

Một cách tiếp cận khác của kỹ thuật tham ăn vào bài toán này là:

1. Xuất phát từ một đỉnh bất kỳ, chọn một cạnh có độ dài nhỏ nhất trong tất cả các cạnh đi ra từ đỉnh đó để đến đỉnh kế tiếp.
2. Từ đỉnh kế tiếp ta lại chọn một cạnh có độ dài nhỏ nhất đi ra từ đỉnh này thoả mãn hai điều kiện nói trên để đi đến dỉnh kế tiếp.

3. Lặp lại bước 2 cho đến khi đi tới đỉnh n thì quay trở về đỉnh xuất phát.

5.2.5. Bài toán cái ba lô

Cho một cái ba lô có thể đựng một trọng lượng W và n loại đồ vật, mỗi đồ vật thứ i có một trọng lượng là g_i và một giá trị là v_i. Tất cả các loại đồ vật đều có số lượng không hạn chế. Tìm một cách lựa chọn các đồ vật đựng vào ba lô, chọn các loại đồ vật nào, mỗi loại lấy bao nhiêu sao cho tổng trọng lượng không vượt quá W và tổng giá trị là lớn nhất.

Theo yêu cầu của bài toán thì ta cần những đồ vật có giá trị cao mà trọng lượng lại nhỏ để sao cho có thể mang được nhiều "đồ quý", sẽ là hợp lý khi ta quan tâm đến yếu tố "đơn giá" của từng loại đồ vật tức là tỷ lệ giá trị/trọng lượng. Đơn giá càng cao thì đồ càng quý. Từ đó ta có kỹ thuật greedy áp dụng cho bài toán này là:

1. Tính đơn giá cho các loại đồ vật.
2. Xét các loại đồ vật theo thứ tự đơn giá từ lớn đến nhỏ.
3. Với mỗi đồ vật được xét sẽ lấy một số lượng tối đa mà trọng lượng còn lại của ba lô cho phép.
4. Xác định trọng luợng còn lại của ba lô và quay lại bước 3 cho đến khi không còn có thể chọn được đồ vật nào nữa.

Ví dụ 5.2: Ta có một ba lô có trọng lượng là 37 và 4 loại đồ vật với trọng lượng và giá trị tương ứng được cho trong bảng bên.

Loại đồ vật	Trọng lượng	Giá trị
A	15	30
B	10	25
C	2	2
D	4	6

Từ bảng đã cho ta tính đơn giá cho các loại đồ vật và sắp xếp các loại đồ vật này theo thứ tự đơn giá giảm dần ta có bảng sau.

Loại đồ vật	Trọng lượng	Giá trị	Đơn giá
B	10	25	2.5
A	15	30	2.0
D	4	6	1.5
C	2	2	1.0

Theo đó thì thứ tự ưu tiên để chọn đồ vật là là B, A, D và cuối cùng là C. Vật B được xét đầu tiên và ta chọn tối đa 3 cái vì mỗi cái vì trọng lượng mỗi cái là 10 và ba lô có trọng lượng 37. Sau khi đã chọn 3 vât loại B, trọng lượng còn lại trong ba lô là 37 - 3*10 = 7. Ta xét đến vật A, vì A có trọng lượng 15 mà trọng lượng còn lại của balô chỉ còn 7 nên không thể chọn vật A. Xét vật D và ta thấy có thể chọn 1 vật D, khi đó trọng lượng còn lại của ba lô là 7-4 = 3. Cuối cùng ta chọn được một vật C.

Như vậy chúng ta đã chọn 3 cái loại B, một cái loại D và 1 cái loại C. Tổng trọng lượng là 3*10 + 1*4 + 1*2 = 36 và tổng giá trị là 3*25+1*6+1*2 = 83.

Chú ý: Có một số biến thể của bài toán cái ba lô như sau:

1. Mỗi đồ vật i chỉ có một số lượng s_i. Với bài toán này khi lựa chọn đồ vật thứ i ta không được lấy một số lượng vượt quá s_i.
2. Mỗi đồ vật chỉ có một cái. Với bài toán này thì với mỗi đồ vật ta chỉ có thể chọn hoặc không chọn.

5.3. QUY HOẠCH ĐỘNG

5.3.1. Nội dung kỹ thuật

Như đã nói, kỹ thuật chia để trị thường dẫn chúng ta tới một giải thuật đệ quy. Trong các giải thuật đó, có thể có một số giải thuật có độ phức tạp thời gian mũ. Tuy nhiên, thường chỉ có một số đa thức các bài toán con, điều đó có nghĩa là chúng ta đã phải giải một số bài toán con nào đó nhiều lần. Để tránh việc giải dư thừa một số bài toán con, chúng ta tạo ra một bảng để lưu trữ kết quả của các bài toán con và khi cần chúng ta sẽ sử dụng kết quả đã được lưu trong bảng mà không cần phải giải lại bài toán đó. Lấp đầy bảng kết quả các bài toán con theo một quy luật nào đó để nhận được kết quả của bài toán ban đầu (cũng đã được lưu trong một số ô nào đó của bảng) được gọi là quy hoạch động (dynamic programming). Trong một số trường hợp, để tiết kiệm ô nhớ, thay vì dùng một bảng, ta chỉ dùng một véctơ.

Có thể tóm tắt giải thuật quy hoạch động như sau:

1. Tạo bảng bằng cách:
   a. Gán giá trị cho một số ô nào đó.
   b. Gán trị cho các ô khác nhờ vào giá trị của các ô trước đó.
2. Tra bảng và xác định kết quả của bài toán ban đầu.

Ưu điểm của phương pháp quy hoạch động là chương trình thực hiện nhanh do không phải tốn thời gian giải lại một bài toán con đã được giải.

Kỹ thuật quy hoạch động có thể vận dụng để giải các bài toán tối ưu, các bài toán có công thức truy hồi.

Phương pháp quy hoạch động sẽ không đem lại hiệu quả trong các trường hợp sau:

- Không tìm được công thức truy hồi.
- Số lượng các bài toán con cần giải quyết và lưu giữ kết quả là rất lớn.
- Sự kết hợp lời giải của các bài toán con chưa chắc cho ta lời giải của bài toán ban đầu.

Sau đây chúng ta sẽ phân tích một số bài toán có thể giải bằng kỹ thuật quy hoạch động.

5.3.2. Bài toán tính số tổ hợp

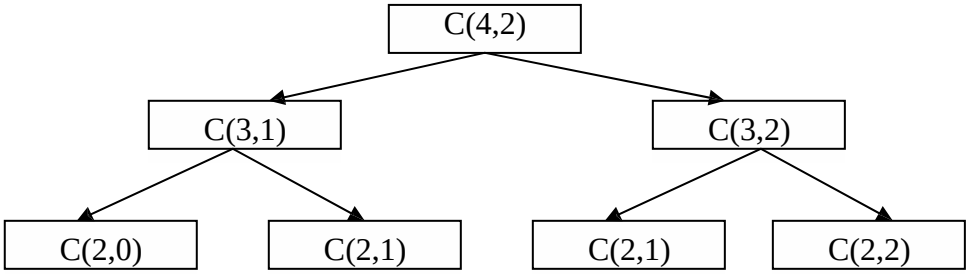
Một bài toán khá quen thuộc là tính số tổ hợp chập k của n theo công thức truy hồi:

$$C_n^k = \begin{cases} 1 & \text{Nếu } k = 0 \text{ hoặc } k = n \\ C_{n-1}^{k-1} + C_{n-1}^{k} & \text{Nếu } 0 < k < n \end{cases}$$

Gọi T(n) là thời gian để tính số tổ hợp chập k của n, thì ta có phương trình đệ quy: T(1) = C1 và T(n) = 2T(n-1) + C2

Giải phương trình này ta được T(n) = O(2n), như vậy là một giải thuật thời gian mũ, trong khi chỉ có một đa thức các bài toán con. Điều đó chứng tỏ rằng có những bài toán con được giải nhiều lần.

Chẳng hạn để tính C(4,2) ta phải tính C(3,1) và C(3,2). Để tính C(3,1) ta phải tính C(2,0) và C(2,1). Để tính C(3,2) ta phải tính Comb(2,1) và C(2,2). Như vậy để tính C(4,2) ta phải tính C(2,1) hai lần. Hình sau minh hoạ rõ điều đó.



Hình 5.5 : Sơ đồ gọi thực hiện C(4,2)

Áp dụng kỹ thuật quy hoạch động để khắc phục tình trạng trên, ta xây dựng một bảng gồm n+1 dòng (từ 0 đến n) và n+1 cột (từ 0 đến n) và điền giá trị cho O(i,j) theo quy tắc sau: (Quy tắc tam giác Pascal):

O(0,0) = 1;

O(i,0) =1;

O(i,i) = 1 với 0 < i (n;

O(i,j) = O(i-1,j-1) + O(i-1,j) với 0 < j < i (n. Chẳng hạn với n = 4 ta có bảng bên.O(n,k) chính là Comb(n,k)

i \ j	0	1	2	3	4
0	1				
1	1	1			
2	1	2	1		
3	1	3	3	1	
4	1	4	6	4	1

5.3.3 Bài toán cái ba lô

Sử dụng kỹ thuật quy hoạch động để giải bài toán cái ba lô đã trình bày trong mục 5.2.5 với một lưu ý là các số liệu đều cho dưới dạng **số nguyên**.

Giả sử X[k,V] là số lượng đồ vật k được chọn, F[k,V] là tổng giá trị của k đồ vật đã được chọn và V là trọng lượng còn lại của ba lô, k = 1..n, V = 1..W.

Trong trường hợp đơn giản nhất, khi chỉ có một đồ vật, ta tính được X[1,V] và F[1,V] với mọi V từ 1 đến W như sau:

X[1,V] = V DIV g1 và F[1,V] = X[1,V] * v1.

Giả sử ta đã tính được F[k-1,V], khi có thêm đồ vật thứ k, ta sẽ tính được F[k,V], với mọi V từ 1 đến W. Cách tính như sau: Nếu ta chọn xk đồ vật loại k, thì trọng lượng còn lại của ba lô dành cho k-1 đồ vật từ 1 đến k-1 là U = V-xk*gk và tổng giá trị của k loại đồ vật đã được chọn F[k,V] = F[k-1,U] + xk*vk, với xk thay đổi từ 0 đến yk= V DIV gk và ta sẽ chọn xk sao cho F[k,V] lớn nhất.

Ta có công thức truy hồi như sau:

X[1,V] = V DIV g1 và F[1,V] = X[1,V] * v1.

F[k,V] = Max(F[k-1,V-xk*gk] + xk*vk) với xk chạy từ 0 đến V DIV gk.

Sau khi xác định được F[k,V] thì X[k,V] là xk ứng với giá trị F[k,V] được chọn trong công thức trên.

Để lưu các giá trị tRung gian trong quá trình tính F[k,V] theo công thức truy hồi trên, ta sử dụng một bảng gồm n dòng từ 1 đến n, dòng thứ k ứng với đồ vật loại k và W+1 cột từ 0 đến W, cột thứ V ứng với trọng lượng V. Mỗi cột V bao gồm hai cột nhỏ, cột bên trái lưu F[k,V], cột bên phải lưu X[k,V]. Trong lập trình ta sẽ tổ chức hai bảng tách rời là F và X.

Ví dụ bài toán cái ba lô với trọng lượng W=9, và 5 loại đồ vật được cho trong bảng sau

Đồ vật	Trọng lượng (gi)	Giá trị (vi)
1	3	4
2	4	5
3	5	6
4	2	3
5	1	1

Ta có bảng F[k,V] và X[k,V] như sau, trong đó mỗi cột V có hai cột con, cột bên trái ghi F[k,V] và cột bên phải ghi X[k,V].

V / k	0		1		2		3		4		5		6		7		8		9	
1	0	0	0	0	0	0	4	1	4	1	4	1	8	2	8	2	8	2	12	3
2	0	0	0	0	0	0	4	0	5	1	5	1	8	0	9	1	10	2	12	0
3	0	0	0	0	0	0	4	0	5	0	6	1	8	0	9	0	10	0	12	0
4	0	0	0	0	3	1	4	0	6	2	7	1	9	3	10	2	12	4	13	3
5	0	0	1	1	3	0	4	0	6	0	7	0	9	0	10	0	12	0	13	0

Trong bảng trên, việc điền giá trị cho dòng 1 rất đơn giản bằng cách sử dụng công thức: X[1,V] = V DIV g1 và F[1,V] = X[1,V] * v1.

Từ dòng 2 đến dòng 5, phải sử dụng công thức truy hồi:

F[k,V] = Max(F[k-1,V-xk*gk] + xk*vk) với xk chạy từ 0 đến V DIV gk.

Ví dụ để tính F[2,7], ta có xk chạy từ 0 đến V DIV gk, trong trường hợp này là xk chạy từ 0 đến 7 DIV 4, tức xk có hai giá trị 0 và 1.

Khi đó F[2,7] = Max (F[2-1, 7-0*4] + 0*5, F[2-1,7-1*4] + 1*5) = Max(F[1,7], F[1,3] + 5) = Max(8, 4+5) = 9. F[2,7] = 9 ứng với xk = 1 do đó X[2,7] = 1.

Vấn đề bây giờ là cần phải tra trong bảng trên để xác định phương án. Khởi đầu, trọng lượng còn lại của ba lô V = W.

Xét các đồ vật từ n đến 1, với mỗi đồ vật k, ứng với trọng lượng còn lại V của ba lô, nếu X[k,V] > 0 thì chọn X[k,V] đồ vật loại k. Tính lại V = V - X[k,V] * gk.

Ví dụ, trong bảng trên, ta sẽ xét các đồ vật từ 5 đến 1. Khởi đầu V = W = 9. Với k = 5, vì X[5,9] = 0 nên ta không chọn đồ vật loại 5.

Với k = 4, vì X[4,9] = 3 nên ta chọn 3 đồ vật loại 4. Tính lại V = 9 – 3 * 2 = 3. Với k = 3, vì X[3,3] = 0 nên ta không chọn đồ vật loại 3.

Với k = 2, vì X[2,3] = 0 nên ta không chọn đồ vật loại 2.

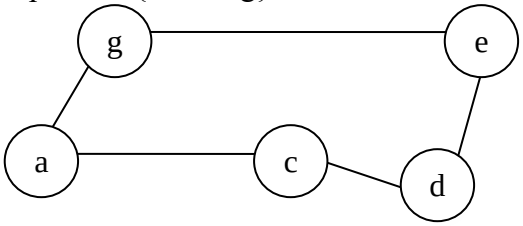
Với k = 1, vì X[1,3] = 1 nên ta chọn 1 đồ vật loại 1. Tính lại V = 3 – 1 * 3 = 0.

Vậy tổng trọng lương các vật được chọn là 3 * 2 + 1 * 3 = 9. Tổng giá trị các vật được chọn là 3 * 3 + 1 * 4 = 13.

5.3.4. Bài toán đường đi của người giao hàng

Chúng ta có thể áp dụng kỹ thuật quy hoạch động để giải bài toán TSP đã trình bày trong mục 5.2.4.

Đặt S = {x_1, x_2, …, x_k} là tập hợp con các cạnh của đồ thị G = (V,E). Ta nói rằng một đường đi P từ v đến w *phủ lên* S nếu P = {v, x_1, x_2, …, x_k, w}, trong đó xi có thể xuất hiện ở một thứ tự bất kì, nhưng chỉ xuất hiện duy nhất một lần. Ví dụ đường cho trong hình sau, đi từ a đến a, phủ lên {c, d, e, g}.



Hình 5.6: Đường đi từ a đến a phủ lên {c, d, e, g}

Ta định nghĩa d(v, w, S) là tổng độ dài của đường đi ngắn nhất từ v đến w, phủ lên S. Nếu không có một đường đi như vậy thì đặt d(v, w, S) = ∞. Một chu trình Hamilton nhỏ nhất Cmin của G phải có tổng độ dài là c(Cmin) = d(v,v, V - {v}). Trong đó v là một đỉnh nào đó của V. Ta xác định Cmin như sau:

Nếu |V| = 1 (G chỉ có một đỉnh) thì c(Cmin) = 0, ngược lại ta có công thức đệ qui để tính d(v, w, S) là: d(v, w, {}) = c(v,w)

d(v, w, S) = min [c(v, x) + d(x, w, S – {x}], lấy với mọi x ∈ S.

Trong đó c(v, w) là độ dài của cạnh nối hai đỉnh v và w nếu nó tồn tại hoặc là ∞ nếu ngược lại. Dòng thứ hai trong công thức đệ qui trên ứng với tập S không rỗng, nó chỉ ra rằng đường đi ngắn nhất từ v đến w phủ lên S, trước hết phải đi đến một đỉnh x nào đó trong S và sau đó là đường đi ngắn nhất từ x đến w, phủ lên tập S – {x}.

Bằng cách lưu trữ các đỉnh x trong công thức đệ qui nói trên, chúng ta sẽ thu được một chu trinh Hamilton tối tiểu.

5.4. KỸ THUẬT QUAY LUI

Kỹ thuật quay lui (backtracking) như tên gọi của nó, là một quá trình phân tích đi xuống và quay lui trở lại theo con đường đã đi qua. Tại mỗi bước phân tích chúng ta chưa giải quyết được vấn đề do còn thiếu cứ liệu nên cứ phải phân tích cho tới các điểm dừng, nơi chúng ta xác định được lời giải của chúng hoặc là xác định được là không thể (hoặc không nên) tiếp tục theo hướng này. Từ các điểm dừng này chúng ta quay ngược trở lại theo con đường mà chúng ta đã đi qua để giải quyết các vấn đề còn tồn đọng và cuối cùng ta sẽ giải quyết được vấn đề ban đầu.

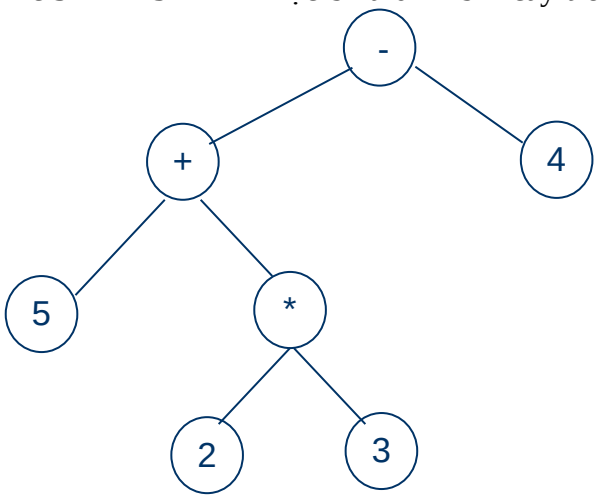
Ở đây chúng ta sẽ xét 3 kỹ thuật quay lui: "vét cạn" là kỹ thuật phải đi tới tất cả các điểm dừng rồi mới quay lui. "Cắt tỉa Alpha-Beta" và "Nhánh-Cận" là hai kỹ thuật cho phép chúng ta không cần thiết phải đi tới tất cả các điểm dừng, mà chỉ cần đi đến một số điểm nào đó và dựa vào một số suy luận để có thể quay lui sớm. Các kỹ thuật này sẽ được trình bày thông qua một số bài toán cụ thể sau.

5.4.1 Định trị cây biểu thức số học

Trong các ngôn ngữ lập trình đều có các biểu thức số học, việc dịch các biểu thức này đòi hỏi phải đánh giá (định trị) chúng. Để làm được điều đó cần phải có một biểu diễn tRung gian cho biểu thức. Một trong các biểu diễn tRung gian cho biểu thức là cây biểu thức.

Cây biểu thức số học là một cây nhị phân, trong đó các nút lá biểu diễn cho các toán hạng, các nút trong biểu diễn cho các toán tử.

Ví dụ 5.3: Biểu thức 5 + 2 * 3 - 4 sẽ được biểu diễn bởi cây trong hình 5.7



Hình 5.7: Một cây biểu thức số học

Trị của một nút lá chính là trị của toán hạng mà nút đó biểu diễn. Trị của một nút trong có được bằng cách lấy toán tử mà nút đó biểu diễn áp dụng vào các con của nó.

Trị của nút gốc chính là trị của biểu thức. Như vậy để định trị cho nút gốc, chúng ta phải định trị cho hai con của nó, đối với mỗi con ta xem nó có phải là nút lá hay không, nếu không phải ta lại phải xét hai con của nút đó. Quá trình cứ tiếp tục như vậy cho tới khi gặp các nút lá mà giá trị của chúng đã được biết, quay lui để định trị cho các nút cha của các nút lá và cứ như thế mà định trị cho tổ tiên của chúng. Đó chính là kỹ thuật quay lui vét cạn, vì chúng ta phải lần đến tất cả các nút lá mới định trị được cho các nút trong và do thế mới định trị được cho nút gốc.

Ví dụ 5.4: Với cây biểu thức trong ví dụ 5-3. Để định trị cho nút - chúng ta phải định trị cho nút + và nút 4. Nút 4 là nút lá nên giá trị của nó là 4. Để định trị cho nút + ta phải định trị cho nút 5 và nút *. Nút 5 là nút lá nên giá trị của nó là 5. Để định trị cho nút *, ta phải định trị cho nút 2 và nút 3. Cả hai nút này đều là lá nên giá trị của chúng tương ứng là 2 và 3. Quay lui lại nút *, lấy toán tử * áp dụng cho hai con của nó là 2 và 3 ta được trị của nút * là 6. Quay lui về nút +, lại áp dụng toán tử + vào hai con của nó là 5 và 6 được trị của nút + là 11. Cuối cùng quay về nút -, áp dụng toán tử - vào hai con của nó là 11 và 4 ta được trị của nút - (nút gốc) là 7. Đó chính là trị của biểu thức. Trong hình 3-9î, mũi tên nét đứt minh họa quá trình đi tìm nút lá và mũi tên nét liền minh họa quá trình quay lui để định trị cho các nút, các số bên phải mỗi nút là trị của nút đó.

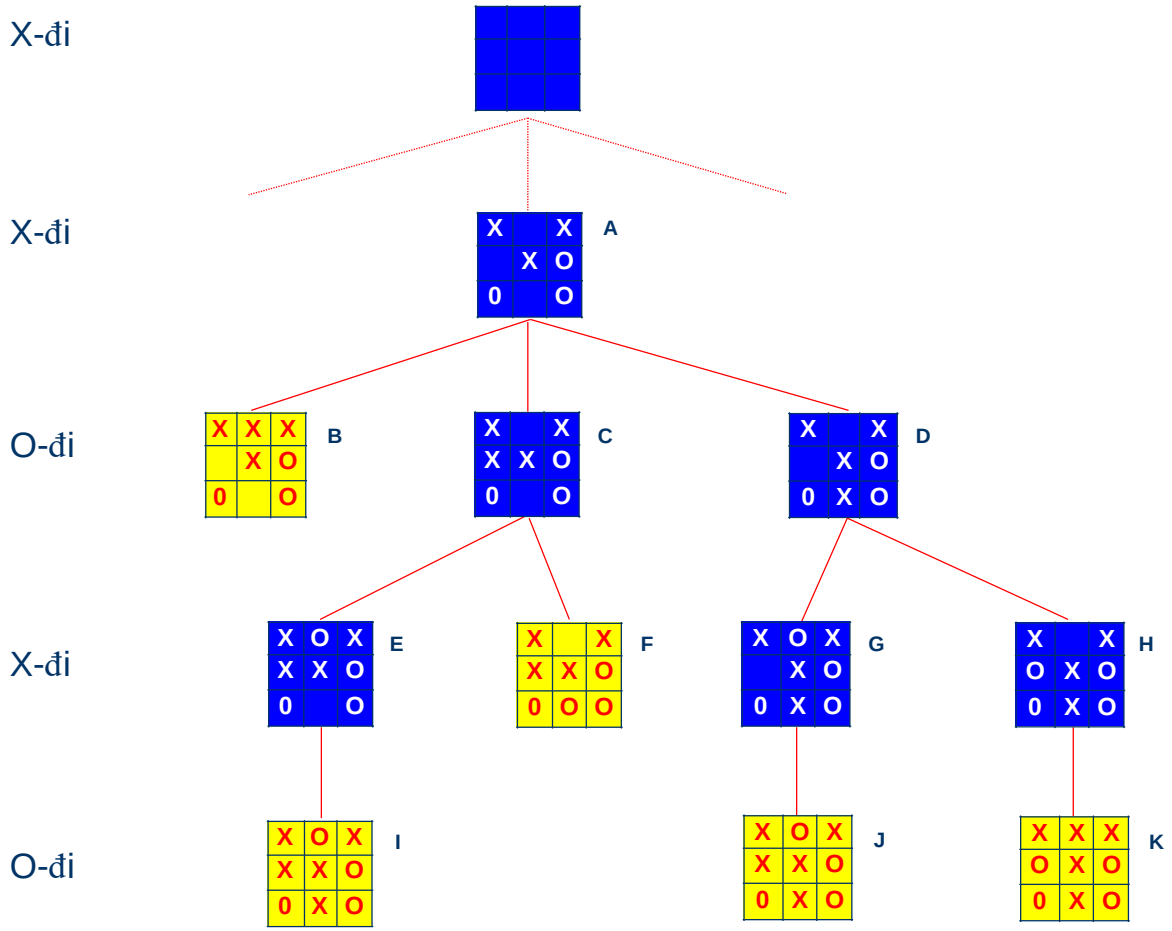
5.4.2 Kỹ thuật cắt tỉa Alpha-Beta

5.4.2.1 Cây trò chơi

Xét một trò chơi trong đó hai người thay phiên nhau đi nước của mình như cờ vua, cờ tướng, carô... Trò chơi có một trạng thái bắt đầu và mỗi nước đi sẽ biến đổi trạng thái hiện hành thành một trạng thái mới. Trò chơi sẽ kết thúc theo một quy định nào đó, theo đó thì cuộc chơi sẽ dẫn đến một trạng thái phản ánh có một người thắng cuộc hoặc một trạng thái mà cả hai đấu thủ không thể phát triển được nước đi của mình, ta gọi nó là trạng thái hòa cờ. Ta tìm cách phân tích xem từ một trạng thái nào đó sẽ dẫn đến đấu thủ nào sẽ thắng với điều kiện cả hai đấu thủ đều có trình độ như nhau.

Một trò chơi như vậy có thể được biểu diễn bởi một cây, gọi là cây trò chơi. Mỗi một nút của cây biểu diễn cho một trạng thái. Nút gốc biểu diễn cho trạng thái bắt đầu của cuộc chơi. Mỗi nút lá biểu diễn cho một trạng thái kết thúc của trò chơi (trạng thái thắng thua hoặc hòa). Nếu trạng thái x được biểu diễn bởi nút n thì các con của n biểu diễn cho tất cả các trạng thái kết quả của các nước đi có thể xuất phát từ trạng thái x.

Ví dụ 5.5: Xét trò chơi carô có 9 ô. Hai người thay phiên nhau đi X hoặc O. Người nào đi được 3 ô thẳng hàng (ngang, dọc, chéo) thì thắng cuộc. Nếu đã hết ô đi mà chưa phân thắng bại thì hai đấu thủ hòa nhau. Một phần của trò chơi này được biểu diễn bởi cây sau:



Hình 5.8: Một phần của cây trò chơi carô 9 ô

Trong cây trò chơi trên, các nút lá được tô nền và viền khung đôi để dễ phân biệt với các nút khác. Ta gắn cho mỗi nút một chữ cái (A, B, C…) để tiện trong việc trình bày các giải thuật.

Ta có thể gán cho mỗi nút lá một giá trị để phản ánh trạng thái thắng thua hay hòa của các đấu thủ. Chẳng hạn ta gán cho nút lá các giá trị như sau:

- 1 nếu tại đó người đi X đã thắng,
- -1 nếu tại đó người đi X đã thua và
- 0 nếu hai đấu thủ đã hòa nhau.

Như vậy từ một trạng thái bất kỳ, đến lượt mình, người đi X sẽ chọn cho mình một nước đi sao cho dẫn đến trạng thái có giá trị lớn nhất (trong trường hợp này là 1). Ta nói X chọn nước đi MAX, nút mà từ đó X chọn nước đi của mình được gọi là nút MAX. Người đi O đến lượt mình sẽ chọn một nước đi sao cho dẫn đến trạng thái có giá trị nhỏ nhất (trong trường hợp này là -1, khi đó X sẽ thua và do đó O sẽ thắng). Ta nói O chọn nước đi MIN, nút mà từ đó O chọn nước đi của mình được gọi là nút MIN. Do hai đấu thủ luân phiên nhau đi nước của mình nên các mức trên cây trò chơi cũng luân phiên nhau là MAX và MIN. Cây trò chơi vì thế còn có tên là cây MIN-MAX. Ta có thể đưa ra một quy tắc định trị cho các nút trên cây để phản ánh tình trạng thắng thua hay hòa và khả năng thắng cuộc của hai đấu thủ.

Nếu một nút là nút lá thì trị của nó là giá trị đã được gán cho nút đó. Ngược lại, nếu nút là nút MAX thì trị của nó bằng giá trị lớn nhất của tất cả các trị của các con của nó. Nếu nút là nút MIN thì trị của nó là giá trị nhỏ nhất của tất cả các trị của các con của nó.

Quy tắc định trị này cũng gần giống với quy tắc định trị cho cây biểu thức số học, điểm khác biệt ở đây là các toán tử là các hàm lấy max hoặc min và mỗi nút có thể có nhiều con. Do vậy ta có thể dùng kỹ thuật quay lui để định trị cho các nút của cây trò chơi.

Ví dụ 5.6: Vận dụng quy tắc quay lui vét cạn để định trị cho nút A trong cây trò chơi trong ví dụ 5.5.

Trước hết ta gán trị cho các nút lá, theo qui định trên thì nút lá B được gán giá trị 1, vì tại đó người đánh X đã thắng. Nút F được gán giá trị -1 vì tại đó người đánh X đã thua (người đánh O đã thắng). Nút I được gán giá trị 0 vì tại đó hai người hòa nhau. Tương tự nút J được gán giá trị 0 và nút K được gán giá trị 1.

Vì người đánh X được gán giá trị 1 tại nút lá mà anh ta đã thắng (giá trị lớn nhất) nên ta nói X chọn nước đi MAX, ngược lại người đánh O sẽ chọn nước đi MIN.

Để định trị cho nút A, ta thấy A là nút MAX và không phải là nút lá nên ta gán giá trị tạm là -∞, xét B là con của A, B là nút lá nên giá trị của nó là giá trị đã được gán 1, giá trị tạm của A bây giờ là max(-∞,1) = 1.

Xét con C của A, C là nút MIN, giá trị tạm lúc đầu của C là ∞.

Xét con E của C, E là nút MAX, giá trị tạm của E là -∞.

Xét con I của E, I là nút lá nên giá trị của nó là 0.

Quay lui lại E, giá trị tạm của E bây giờ là max(-∞,0) = 0. Vì E chỉ có một con là I đã xét nên giá trị tạm 0 trở thành giá trị của E.

Quay lui lại C, giá trị tạm mới của C là min(∞,0) = 0. Lại xét con F của C, vì F là nút lá, nên giá trị của F đã được gán là –1.

Quay lui lại C, giá trị tạm mới của C là min(0,-1) = -1. Nút C có hai con là E và F, cả hai con này đều đã được xét, vậy giá trị tạm -1 của C trở thành giá trị của nó.

Sau khi có giá trị của C, ta phải quay lại A và đặt lại giá trị tạm của A là max(1,-1) = 1.

Tiếp tục xét nút D, D là nút MIN nên giá trị tạm là ∞, xét nút con G của D, G là nút MAX nên giá trị tạm của nó là -∞, xét nút con J của G. Vì J là nút lá nên có giá trị 0.

Quay lui lại G, giá trị tạm của G bây giờ là max(-∞,0) = 0 và giá trị tạm này trở thành giá trị của G vì G chỉ có một con J đã xét.

Quay lui về D, giá trị tạm của D bây giờ là min(∞,0) = 0. Lại xét con H của D, H là nút MAX nên gán giá trị tạm ban đầu là -∞.

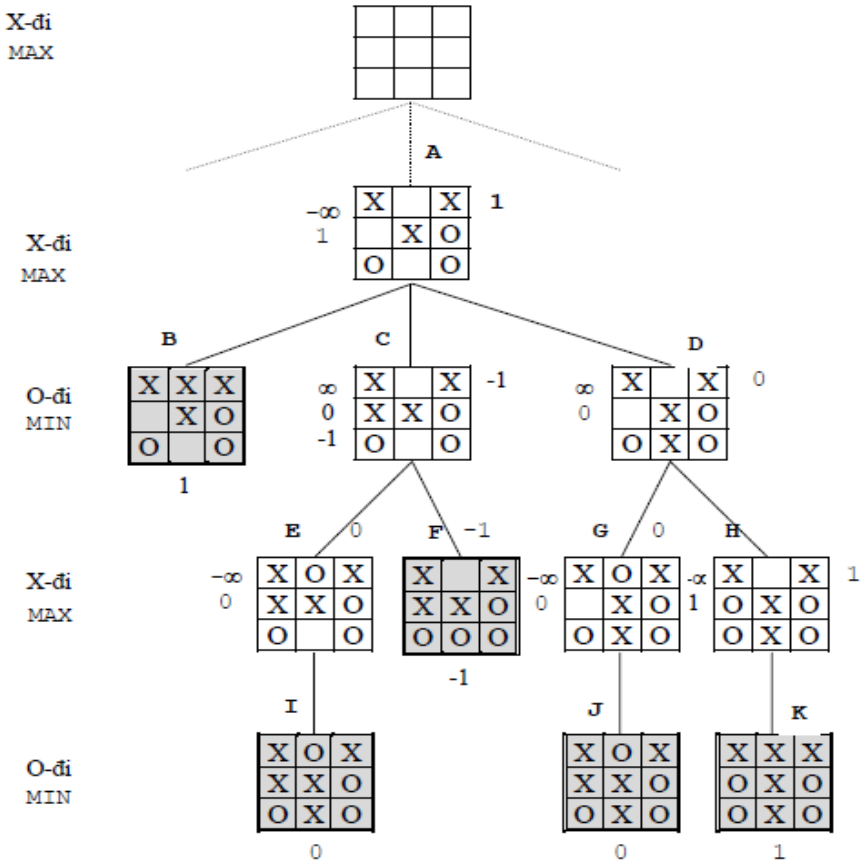
Xét con K của H, nút K là nút lá nên giá trị của K đã được gán là 1.

Quay lui về H và đặt lại giá trị tạm của H là max(-∞,1) = 1. Giá trị tạm này chính là giá trị của H vì H chỉ có một con K đã được xét.

Quay lui về D và đặt lại giá trị tạm của D là min(0, 1) = 0. Cả hai con G và H của D đều đã được xét nên giá trị tạm 0 của D trở thành giá trị của nó.

Quay lui về A, giá trị tạm của nó là max(1,0) = 1vẫn không thay đổi, nhưng lúc này cả 3 con của A đều đã được xét nên giá trị tạm 1 trở thành giá trị của A.

Kết quả được minh họa trong hình sau:



Hình 5.9: Định trị cây trò chơi bằng kỹ thuật quay lui vét cạn

Trong hình trên, các nút lá có giá trị được gán ghi phía dưới mỗi nút. Đối với các nút trong, bên trái ghi các giá trị tạm theo thứ tự trên xuống, các giá trị thực được ghi bên phải hoặc phía trên bên phải.

5.4.2.2 Giải thuật vét cạn định trị cây trò chơi

Để cài đặt ta có một số giả thiết sau:

• Ta có một hàm Payoff nhận vào một nút lá và cho ta giá trị của nút lá đó.

• Các hằng ∞ và -∞ tương ứng là các trị Payoff lớn nhất và nhỏ nhất.

• Khai báo kiểu ModeType = (MIN, MAX) để xác định định trị cho nút là MIN hay MAX.

• Một kiểu NodeType được khai báo một cách thích hợp để biểu diễn cho một nút trên cây phản ánh một trạng thái của cuộc chơi.

• Ta có một hàm is_leaf để xác định xem một nút có phải là nút lá hay không?

• Hàm max và min tương ứng lấy giá trị lớn nhất và giá trị nhỏ nhất của hai giá trị.

Hàm Search nhận vào một nút n và kiểu mode của nút đó (MIN hay MAX) trả về giá trị của nút. Nếu nút n là nút lá thì trả về giá trị đã được gán cho nút lá. Ngược lại ta cho n một giá trị tạm value là -∞ hoặc ∞ tùy thuộc n là nút MAX hay MIN và xét con của n. Sau khi một con của n có giá trị V thì đặt lại value = max(value,V) nếu n là nút MAX và value = min(value,V) nếu n là nút MIN. Khi tất cả các con của n đã được xét thì giá trị tạm value của n trở thành giá trị của nó.

5.4.2.3 Kỹ thuật cắt tỉa Alpha-Beta (Alpha-Beta PRuning)

Trong giải thuật vét cạn ở trên, ta thấy để định trị cho một nút nào đó, ta phải định trị cho tất cả các nút con cháu của nó, và muốn định trị cho nút gốc ta phải định trị cho tất cả các nút trên cây. Số lượng các nút trên cây trò chơi tuy hữu hạn nhưng không phải là ít. Chẳng hạn trong cây trò chơi ca rô nói trên, nếu ta có bàn cờ bao gồm n ô thì có thể có tới n! nút trên

cây (trong trường hợp trên là 9!). Đối với các loại cờ khác như cờ vua chẳng hạn, thì số lượng các nút còn lớn hơn nhiều. Ta gọi là một sự bùng nổ tổ hợp các nút.

Chúng ta cố gắng tìm một cách sao cho khi định trị một nút thì không nhất thiết phải định trị cho tất cả các nút con cháu của nó. Trước hết ta có nhận xét như sau:

Nếu P là một nút MAX và ta đang xét một nút con Q của nó (dĩ nhiên Q là nút MIN). Giả sử Vp là một giá trị tạm của P, Vq là một giá trị tạm của Q và nếu ta có Vp ≥ Vq thì ta không cần xét các con chưa xét của Q nữa. Vì nếu có xét thì giá trị của Q cũng sẽ nhỏ hơn hoặc bằng Vq và do đó không ảnh hưởng gì đến Vp. Tương tự nếu P là nút MIN (tất nhiên Q là nút MAX) và Vp ≤ Vq thì ta cũng không cần xét đến các con chưa xét của Q nữa. Việc không xét tiếp các con chưa được xét của nút Q gọi là việc cắt tỉa Alpha-Beta các con của nút Q.

Trên cơ sở nhận xét đó, ta nêu ra quy tắc định trị cho một nút không phải là nút lá trên cây như sau:

1. Khởi đầu nút MAX có giá trị tạm là -∞ và nút MIN có giá trị tạm là ∞.

2. Nếu tất cả các nút con của một nút đã được xét hoặc bị cắt tỉa thì giá trị tạm của nút đó trở thành giá trị của nó.

3. Nếu một nút MAX n có giá trị tạm là V1 và một nút con của nó có giá trị là V2 thì đặt giá trị tạm mới của n là max(V1,V2). Nếu n là nút MIN thì đặt giá trị tạm mới của n là min(V1,V2).

4. Vận dụng quy tắc cắt tỉa Alpha-Beta nói trên để hạn chế số lượng nút phải xét.

Ví dụ 5-7: Vận dụng quy tắc trên để định trị cho nút A của cây trò chơi trong ví dụ 5-5. A là nút MAX, vì A không phải là nút lá nên ta gán giá trị tạm là -∞, xét B là con của A, B là nút lá nên giá trị của nó là giá trị đã được gán 1, giá trị tạm của A bây giờ là max(-∞,1) = 1. Xét con C của A, C là nút MIN, giá trị tạm lúc đầu của C là ∞.

Xét con E của C, E là nút MAX, giá trị tạm của E là -∞. Xét con I của E, I là nút lá nên giá trị của nó là 0.

Quay lui lại E, giá trị tạm của E bây giờ là max(-∞,0) = 0. Vì E chỉ có một con là I đã xét nên giá trị tạm 0 trở thành giá trị của E.

Quay lui lại C, giá trị tạm mới của C là min(∞,0) = 0. A là nút MAX có giá trị tạm là 1, C là con của A, có giá trị tạm là 0, 1>0 nên ta không cần xét con F của C nữa. Nút C có hai con là E và F, trong đó E đã được xét, F đã bị cắt, vậy giá trị tạm 0 của C trở thành giá trị của nó.

Sau khi có giá trị của C, ta phải đặt lại giá trị tạm của A, nhưng giá trị tạm này không thay đổi vì max(1,0) = 1.

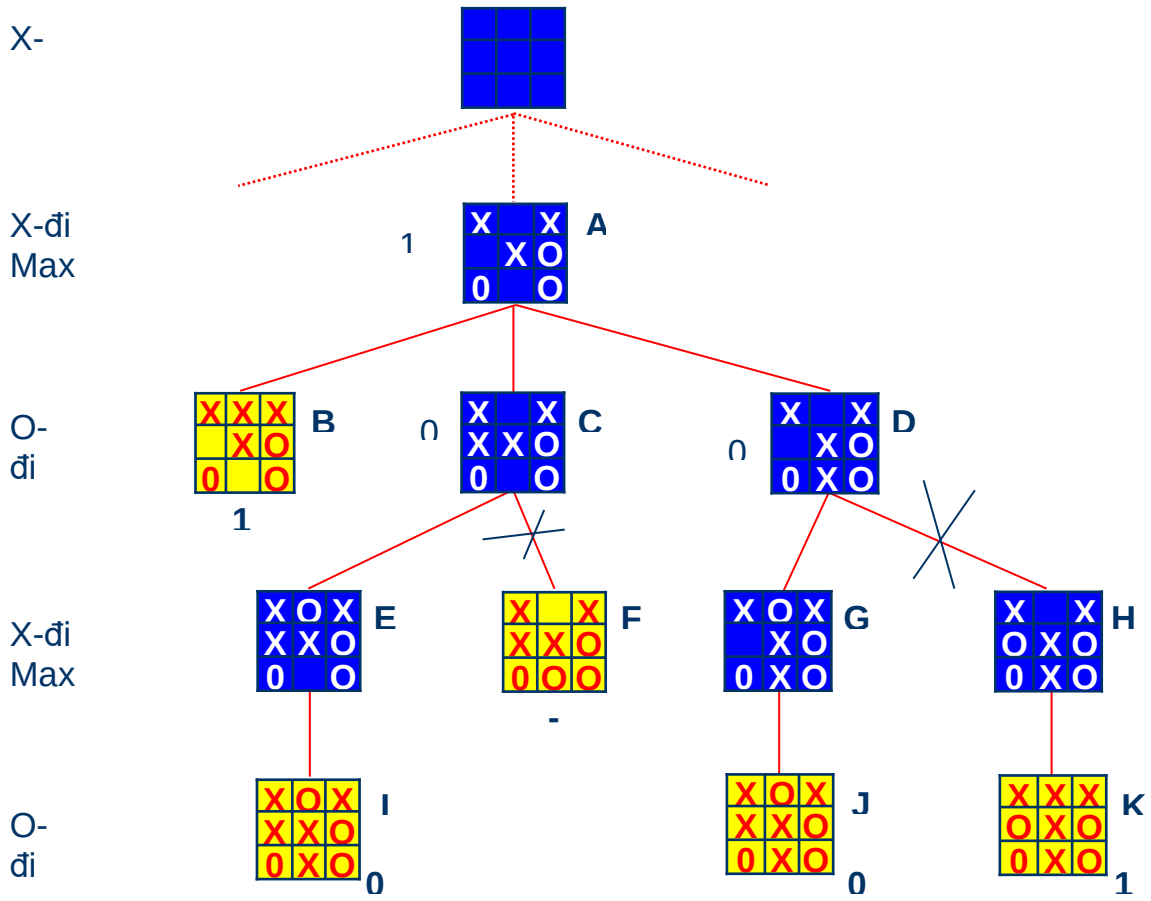
Tiếp tục xét nút D, D là nút MIN nên giá trị tạm là ∞, xét nút con G của D, G là nút MAX nên giá trị tạm của nó là -∞, xét nút con J của G. Vì J là nút lá nên có giá trị 0.

Quay lui lại G, giá trị tạm của G bây giờ là max(-∞,0) = 0 và giá trị tạm này trở thành giá trị của G vì G chỉ có một con J đã xét.

Quay lui về D, giá trị tạm của D bây giờ là min(∞,0) = 0. Giá trị tạm này của D nhỏ hơn giá trị tạm của nút A MAX là cha của nó nên ta cắt tỉa con H chưa được xét của D và lúc này D có giá trị là 0.

Quay lui về A, giá trị tạm của nó vẫn không thay đổi, nhưng lúc này cả 3 con của A đều đã được xét nên giá trị tạm 1 trở thành giá trị của A.

Kết quả được minh họa trong hình sau:



Hình 5.10: Định trị cây trò chơi bằng kỹ thuật cắt tỉa alpha-beta

5.4.3 Kỹ thuật nhánh cận

Với các bài toán tìm phương án tối ưu, nếu chúng ta xét hết tất cả các phương án thì mất rất nhiều thời gian, nhưng nếu sử dụng phương pháp tham ăn thì phương án tìm được chưa hẳn đã là phương án tối ưu. Nhánh cận là kỹ thuật xây dựng cây tìm kiếm phương án tối ưu, nhưng không xây dựng toàn bộ cây mà sử dụng giá trị cận để hạn chế bớt các nhánh.

Cây tìm kiếm phương án có nút gốc biểu diễn cho tập tất cả các phương án có thể có, mỗi nút lá biểu diễn cho một phương án nào đó. Nút n có các nút con tương ứng với các khả năng có thể lựa chọn tập phương án xuất phát từ n. Kỹ thuật này gọi là phân nhánh.

Với mỗi nút trên cây ta sẽ xác định một giá trị cận. Giá trị cận là một giá trị gần với giá của các phương án. Với bài toán tìm min ta sẽ xác định cận dưới còn với bài toán tìm max ta sẽ xác định cận trên. Cận dưới là giá trị nhỏ hơn hoặc bằng giá của phương án, ngược lại cận trên là giá trị lớn hơn hoặc bằng giá của phương án.

Để dễ hình dung ta sẽ xét hai bài toán quen thuộc là bài toán TSP và bài toán cái ba lô.

5.4.3.1 Bài toán đường đi của người giao hàng

5.4.3.1.1 Phân nhánh

Cây tìm kiếm phương án là cây nhị phân trong đó:

• Nút gốc là nút biểu diễn cho cấu hình bao gồm tất cả các phương án.

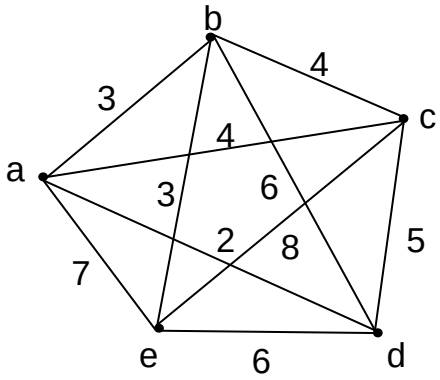
• Mỗi nút sẽ có hai con, con trái biểu diễn cho cấu hình bao gồm tất cả các phương án chứa một cạnh nào đó, con phải biểu diễn cho cấu hình bao gồm tất cả các phương án không chứa cạnh đó (các cạnh để xét phân nhánh được thành lập tuân theo một thứ tự nào đó, chẳng hạn thứ tự từ điển).

• Mỗi nút sẽ kế thừa các thuộc tính của tổ tiên của nó và có thêm một thuộc tính mới (chứa hay không chứa một cạnh nào đó).

• Nút lá biểu diễn cho một cấu hình chỉ bao gồm một phương án.

• Để quá trình phân nhánh mau chóng tới nút lá, tại mỗi nút ta cần có mộtquyết định bổ sung dựa trên nguyên tắc là mọi đỉnh trong chu trình đềucó cấp 2 và không tạo ra một chu trình thiếu.

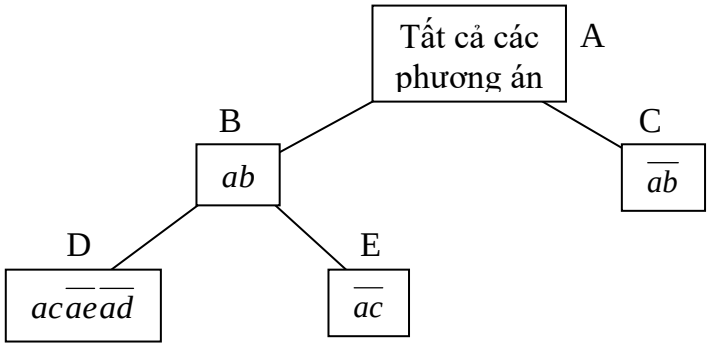
Ví dụ 5-7: Xét bài toán TSP có 5 đỉnh với độ dài các cạnh được cho trong hình 5- 11.



Hình 5.11: Bài toán TSP có 5 đỉnh

Các cạnh theo thứ tự từ điển để xét là: ab, ac, ad, ae, bc, bd, be, cd, ce và de. Nút gốc A của cây bao gồm tất cả các phương án. Hai con của A là B và C, trong đó B bao gồm tất cả các phương án chứa cạnh ab, C bao gồm tất cả các phương án không chứa ab, kí hiệu là $\overline{ab}$. Hai con của B là D và E. Nút D bao gồm tất cả các phương án chứa ac. Vì các phương án này vừa chứa ab (kế thừa của B) vừa chứa ac nên đỉnh a đã đủ cấp hai nên D không thể chứa ad và ae. Nút E bao gồm tất cả các phương án không chứa ac…

Ta được cây (chưa đầy đủ) trong hình 5.12.



Hình 5.12: Phân nhánh

5.4.3.1.2 Tính cận dưới

Đây là bài toán tìm min nên ta sử dụng cận dưới. Cận dưới tại mỗi nút là một số nhỏ hơn hoặc bằng giá của tất cả các phương án được biểu diễn bởi nút đó. Giá của một phương án ở đây là tổng độ dài của một chu trình. Để tính cận dưới của nút gốc, mỗi đỉnh ta chọn hai cạnh có độ dài nhỏ nhất. Cận dưới của nút gốc bằng tổng độ dài tất cả các cạnh được chọn chia cho 2.

Ví dụ 5.8: Với số liệu cho trong ví dụ 5-7 nói trên, ta tính cận dưới của nút gốc A (hình 5-12) như sau:

• Đỉnh a chọn ad = 2, ab = 3

• Đỉnh b chọn ba = 3, be = 3

• Đỉnh c chọn ca = 4, cb = 4

• Đỉnh d chọn da = 2, dc = 5

• Đỉnh e chọn eb = 3, ed = 6

Tổng độ dài các cạnh được chọn là 35, cận dưới của nút gốc A là 35/2 = 17.5

Đối với các nút khác, chúng ta phải lựa chọn hai cạnh có độ dài nhỏ nhất thỏa điều kiện ràng buộc (phải chứa cạnh này, không chứa cạnh kia).

Ví dụ 5.9: Tính cận dưới cho nút D trong hình 5-12. Điều kiện ràng buộc là phải chứa ab, ac và không chứa ad, ae.

- Đỉnh a chọn ab = 3, ac = 4, do hai cạnh này buộc phải chọn.
- Đỉnh b chọn ba = 3, be = 3
- Đỉnh c chọn ca = 4, cb = 4
- Đỉnh d chọn de = 6, dc = 5, do không được chọn da nên ta phải chọn de.
- Đỉnh e chọn eb = 3, ed = 6

Tổng độ dài các cạnh được chọn là 41, cận dưới của nút D là 41/2 = 20.5

5.4.3.1.3 Kỹ thuật nhánh cận

Bây giờ ta sẽ kết hợp hai kỹ thuật trên để xây dựng cây tìm kiếm phương án. Quy tắc như sau:

- Xây dựng nút gốc, bao gồm tất cả các phương án, tính cận dưới cho nút gốc.
- Sau khi phân nhánh cho mỗi nút, ta tính cận dưới cho cả hai con.
- Nếu cận dưới của một nút con lớn hơn hoặc bằng giá nhỏ nhất tạm thời của một phương án đã được tìm thấy thì ta không cần xây dựng các cây con cho nút này nữa (Ta gọi là cắt tỉa các cây con của nút đó).
- Nếu cả hai con đều có cận dưới nhỏ hơn giá nhỏ nhất tạm thời của mộtphương án đã được tìm thấy thì nút con nào có cận dưới nhỏ hơn sẽ được ưu tiên phân nhánh trước.
- Mỗi lần quay lui để xét nút con chưa được xét của một nút ta phải xem xét lại nút con đó để có thể cắt tỉa các cây của nó hay không vì có thể một phương án có giá nhỏ nhất tạm thời vừa được tìm thấy.
- Sau khi tất cả các con đã được phân nhánh hoặc bị cắt tỉa thì phương án có giá nhỏ nhất trong các phương án tìm được là phương án cần tìm.

Trong quá trình xây dựng cây có thể ta đã xây dựng được một số nút lá, như ta biết mỗi nút lá biểu diễn cho một phương án. Giá nhỏ nhất trong số các giá của các phương án này được gọi là giá nhỏ nhất tạm thời.

Ví dụ 5.10: Xét bài toán TSP trong ví dụ 5-7 nói trên.

Tập hợp các cạnh để xét phân nhánh là ab, ac, ad, ae, bc, bd, be, cd, ce và de. Điều kiện bổ sung ở đây là mỗi đỉnh phải được chọn hai cạnh, bị loại hai cạnh và không được tạo ra chu trình thiếu.

Nút gốc A bao gồm tất cả các phương án, có cận dưới là 17.5. Phân nhánh cho A, xây dựng hai con là B và C. Tính cận dưới cho hai nút này được cận dưới của B là 17.5 và C là 18.5. Nút B có cận dưới nhỏ hơn nên được phân nhánh trước. Hai con của B là D và E. Các ràng buộc của D và E giống nh-ư ta đã nói trong ví dụ của phần phân nhánh. Tính cận cho D và E, được cận dưới của D là 20.5 và của E là 18. Nút E được xét trước. Phân nhánh cho nút E theo cạnh ad, hai con của E là F và G. F chứa ad và G không chứa ad. Do F kế thừa các thuộc tính của E và B, nên F là tập hợp các phương án chứa ab, ad và không chứa ac, đỉnh a đã đủ cấp 2 vậy F không chứa ae. Tương tự G chứa ab, không chứa ac, không chứa ad nên phải chứa ae. Tính cận dưới cho F và G được cận dưới của F là 18 và của G là 23. Tiếp tục xây dựng hai con cho F theo cạnh bc là H và I. H chứa bc và I không chứa bc. Do H kế thừa các thuộc tính của B, E và F nên H là các phương án chứa ab, ad, không chứa ac và chứa bc. Như vậy đỉnh a đã thỏa điều kiện là được chọn hai cạnh (ab và ad) và bị loại hai cạnh (ac và ae), Đỉnh b đã được chọn 2 cạnh (ba và bc) nên bd và be bị loại.

Đỉnh c đã được chọn cb, bị loại ca, ta có thể chọn cd hoặc ce. Nếu chọn cd thì sẽ có một chu trinh thiếu a b c d a, như vậy cd bị loại nên phải chọn ce. Đỉnh d có db và dc đã bị loại, da đã được chọn nên phải chọn thêm de. Lúc đó đỉnh e cũng đã có hai cạnh được chọn là ec và ed, hai cạnh bị loại là eb và ea. Tóm lại H là tập chỉ bao gồm một phương án a b c e d a có

giá là 23. Đối với I ta đã có I chứa ab, không chứa ac, chứa ad, không chứa ae và không chứa bc. Bằng lý luận tương tự ta có I không chứa bd, chứa be, cd, ce và không chứa de. Một phương án mới là a b e c d a với giá 21. Đây là giá nhỏ nhất tạm thời mới được tìm thấy.

Bây giờ ta quay lui về E và xét nút con của nó là G. Vì G có cận dưới là 23 lớn hơn giá thấp nhất tạm thời 21 nên cắt tỉa các con của G.

Quay lui về B và xét nút con D của nó. Cận dưới của D là 20.5 không lớn hơn 21. Nhưng vì độ dài các cạnh trong bài toán đã cho là số nguyên nên nếu ta triển khai các con của D tới nút lá gồm một phương án. Giá của phương án này phải là một số nguyên lớn hơn 20.5 hay lớn hơn hoặc bằng 21. Vậy ta cũng không cần xây dựng các con của D nữa.

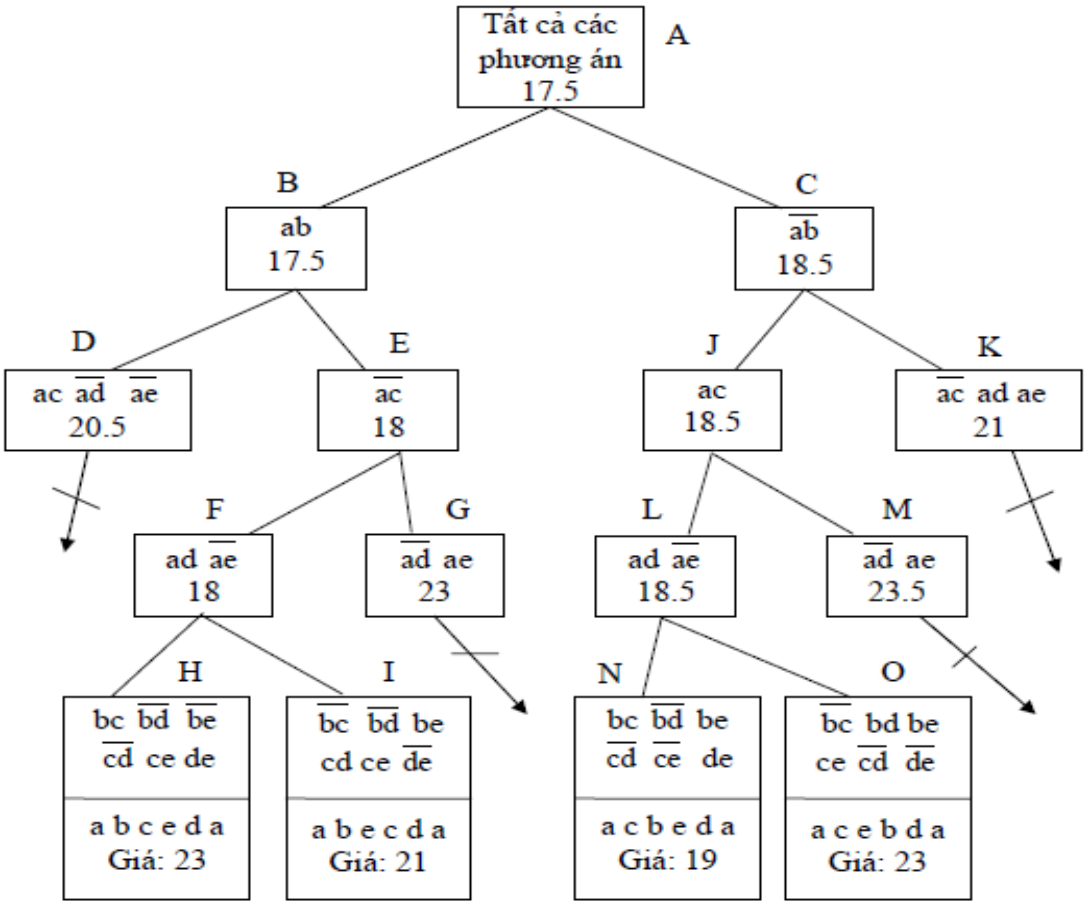
Tiếp tục quay lui đến A và xét con C của nó. Phân nhánh C theo cạnh ac thành hai con J và K. J chứa ac có cận dưới là 18.5. K không chứa ac nên phải chứa ad và ae, cận dưới của K là 21 bằng giá nhỏ nhất tạm thời nên cắt tỉa các con của K.

Hai con của J là L và M. M không chứa ad, ab, chứa ac và ae có cận dưới 23.5 nên bị cắt tỉa các con. Hai con của L là N và O, N chứa bc và O không chứa bc.

Xét nút N ta có: Đỉnh a được chọn hai cạnh ac và ad, bị loại hai cạnh ab và ae. Đỉnh b đã được chọn bc, bị loại ba, ta có thể chọn bd hoặc be. Nếu chọn bd thì sẽ có một chu trình thiếu là a c b d a, vậy phải loại bd và chọn be. Đỉnh c đã được chọn ca, cb nên phải loại cd và ce. Đỉnh d đã được chọn da, bị loại db và dc nên phải chọn de. Khi đó đỉnh e có đủ hai cạnh được chọn là eb, ed và hai cạnh bị loại là ea và ec. Vậy N bao gồm chỉ một phương án là a c b e d a với giá 19.

Tương tự nút O bao gồm chỉ một phương án a c e b d a có giá là 23.

Tất cả các nút con của cây đã được xét hoặc bị cắt tỉa nên phương án cần tìm là a c b e d a với giá 19. Hình 5.13 minh họa cho những điều ta vừa nói.



Hình 5.13: Kỹ thuật nhánh cận giải bài toán TSP

5.4.3.2 Bài toán cái ba lô

Ta thấy đây là một bài toán tìm max. Danh sách các đồ vật được sắp xếp theo thứ tự giảm của đơn giá để xét phân nhánh.

1. Nút gốc biểu diễn cho trạng thái ban đầu của ba lô, ở đó ta chưa chọn một vật nào. Tổng giá trị được chọn TGT = 0. Cận trên của nút gốc CT = W * Đơn giá lớn nhất.

2. Nút gốc sẽ có các nút con tương ứng với các khả năng chọn đồ vật có đơn giá lớn nhất. Với mỗi nút con ta tính lại các thông số:

- TGT = TGT (của nút cha) + số đồ vật được chọn * giá trị mỗi vật.
- W = W (của nút cha) - số đồ vật được chọn * trọng lượng mỗi vật.
- CT = TGT + W * Đơn giá của vật sẽ xét kế tiếp.

3. Trong các nút con, ta sẽ ưu tiên phân nhánh cho nút con nào có cận trên lớn hơn trước. Các con của nút này tương ứng với các khả năng chọn đồ vật có đơn giá lớn tiếp theo. Với mỗi nút ta lại phải xác định lại các thông số TGT, W, CT theo công thức đã nói trong bước 2.

4. Lặp lại bước 3 với chú ý: đối với những nút có cận trên nhỏ hơn hoặc bằng giá lớn nhất tạm thời của một phương án đã được tìm thấy thì ta không cần phân nhánh cho nút đó nữa (cắt bỏ).

5. Nếu tất cả các nút đều đã được phân nhánh hoặc bị cắt bỏ thì phương án có giá lớn nhất là phương án cần tìm.

Ví dụ 5.11: Với bài toán cái ba lô đã cho trong ví dụ 3-2 , sau khi tính đơn giá cho các đồ vật và sắp xếp các đồ vật theo thứ tự giảm dần của đơn giá ta được bảng sau.

Loại đồ vật	Trọng lượng	Giá trị	Đơn giá
b	10	25	2.5
a	15	30	2.0
d	4	6	1.5
c	2	2	1

Gọi XA, XB, XC, XD là số lượng cần chọn tương ứng của các đồ vật a, b, c d.

Nút gốc A biểu diễn cho trạng thái ta chưa chọn bất cứ một đồ vật nào. Khi đó tổng giá trị TGT =0, trọng lượng của ba lô W=37 (theo đề ra) và cận trên CT = 37*2.5 =92.5, trong đó 37 là W, 2.5 là đơn giá của đồ vật b.

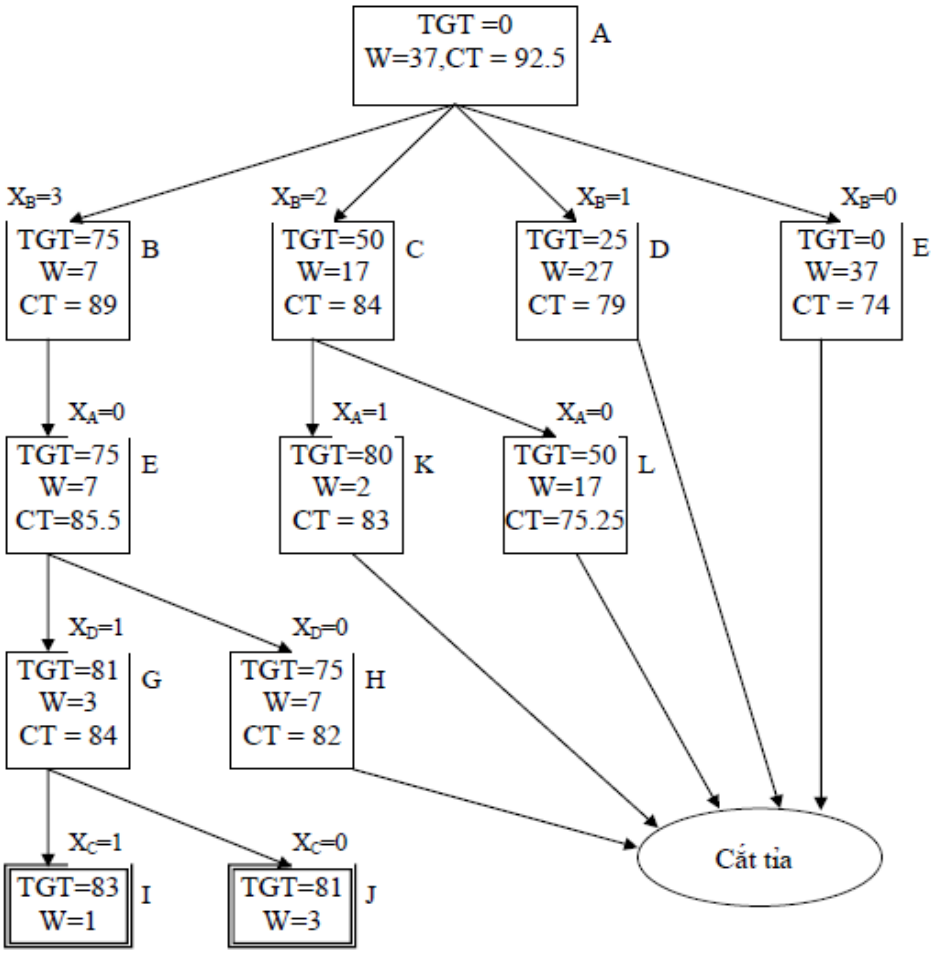
Với đồ vật b, ta có 4 khả năng: chọn 3 đồ vật b (XB=3), chọn 2 đồ vật b (XB=2), chọn 1 đồ vật b (XB=1) và không chọn đồ vật b (XB=0). Ứng với 4 khả năng này, ta phân nhánh cho nút gốc A thành 4 con B, C, D và E.

Với nút con B, ta có TGT = 0+ 3*25 = 75, trong đó 3 là số vật b được chọn, 25 là giá trị của mỗi đồ vật b. W = 37- 3*10 = 7, trong đó 37 là trọnh lượng ban đầu của ba lô, 3 là số vật b được, 10 là trọng lượng mõi đồ vật b. CT = 75 + 7*2 = 89, trong đó 75 là TGT, 7 là trọng lượng còn lại của ba lô và 2 là đơn giá của đồ vật a. Tương tự ta tính được các thông số cho các nút C, D và E, trong đó cận trên tương ứng là 84, 79 và 74.

Trong các nút B, C, D và E thì nút B có cận trên lớn nhất nên ta sẽ phân nhánh cho nút B trước với hy vọng sẽ có được phương án tốt từ hướng này. Từ nút B ta chỉ có một nút con F duy nhất ứng với XA=0 (do trọng lượng còn lại của ba lô là 7, trong khi trọng lượng của mỗi đồ vật a là 15). Sau khi xác định các thông số cho nút F ta có cận trên của F là 85.5. Ta tiếp tục phân nhánh cho nút F. Nút F có 2 con G và H tương ứng với XD=1 và XD=0. Sau khi xác định các thông số cho hai nút này ta thấy cận trên của G là 84 và của H là 82 nên ta tiếp tục phân nhánh cho nút G. Nút G có hai con là I và J tương ứng với XC=1 và XC=0. Đây là hai nút lá (biểu diễn cho phương án) vì với mỗi nút thì số các đồ vật đã được chọn xong. Trong đó nút I biểu diễn cho phương án chọn XB=3, XA=0, XD=1 và XC=1 với giá 83, trong khi nút J biểu diễn cho phương án chọn XB=3, XA=0, XD=1 và XC=0 với giá 81. Như vậy giá lớn nhất tạm thời ở đây là 83.

Quay lui lên nút H, ta thấy cận trên của H là 82<83 nên cắt tỉa nút H.

Quay lui lên nút C, ta thấy cận trên của C là 84>83 nên tiếp tục phân nhánh cho nút C. Nút C có hai con là K và L ứng với XA=1 và XA=0. Sau khi tính các thông số cho K và L ta thấy cận trên của K là 83 và của L là 75.25. Cả hai giá trị này đều không lớn hơn 83 nên cả hai nút này đều bị cắt tỉa. Cuối cùng các nút D và E cũng bị cắt tỉa. Như vậy tất cả các nút trên cây đều đã được phân nhánh hoặc bị cắt tỉa nên phương án tốt nhất tạm thời là phương án cần tìm. Theo đó ta cần chọn 3 đồ vật loại b, 1 đồ vật loạ d và một đồ vật loại c với tổng giá trị là 83, tổng trọng lượng là 36. Xem minh hoạ trong hình 5.14.



Hình 5.14: Kỹ thuật nhánh cận áp dụng cho bài toán cái ba lô

5.5. KỸ THUẬT TÌM KIẾM ĐỊA PHƯƠNG

5.5.1. Nội dung kỹ thuật

Kỹ thuật tìm kiếm địa phương (local search) thường được áp dụng để giải các bài toán tìm lời giải tối ưu. Phương pháp như sau:

- Xuất phát từ một phương án nào đó.
- Áp dụng một phép biến đổi lên phương án hiện hành để được một phương án mới tốt hơn phương án đã có.
- Lặp lại việc áp dụng phép biến đổi lên phương án hiện hành cho đến khi không còn có thể cải thiện được phương án nữa.

Thông thường một phép biến đổi chỉ thay đổi một bộ phận nào đó của phương án hiện hành để được một phương án mới nên phép biến đổi được gọi là phép biến đổi địa phương và do đó ta có tên kỹ thuật tìm kiếm địa phương. Sau đây ta sẽ trình bày một số ví dụ áp dụng kỹ thuật tìm kiếm địa phương.

5.5.2 Bài toán cây phủ tối thiểu

Cho G = (V,E) là một đồ thị vô hướng liên thông, trong đó V là tập các đỉnh và E là tập các cạnh. Các cạnh của đồ thị G đều có trọng số. Cây T có tập hợp các nút là V được gọi là cây phủ (spaning tree) của đồ thị G.

Cây phủ tối thiểu là một cây phủ của G mà tổng độ dài (trọng số) các cạnh nhỏ nhất. Bài toán cây phủ tối thiểu thường được áp dụng trong việc thiết kế một mạng lưới giao thông giữa các thành phố hay thiết kế một mạng máy tính.

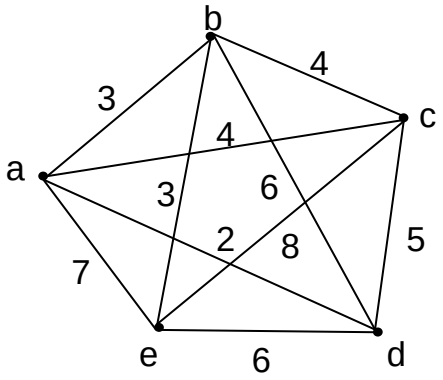
Kỹ thuật tìm kiếm địa phương áp dụng vào bài toán này như sau:

• Phương án ban đầu là một cây phủ nào đó.

• Thành lập tập tất cả các cạnh theo thứ tăng dần của độ dài (có $\frac{n(n-1)}{2}$ cạnh đối với đồ thị có *n* đỉnh).

• Phép biến đổi địa phương ở đây là: Chọn một cạnh có độ dài nhỏ nhất trong tập các cạnh chưa sử dụng để thêm vào cây. Trong cây sẽ có một chu trình, loại khỏi chu trình cạnh có độ dài lớn nhất trong chu trình đó. Ta được một cây phủ mới. Lặp lại bước này cho đến khi không còn cải thiện được phương án nữa.

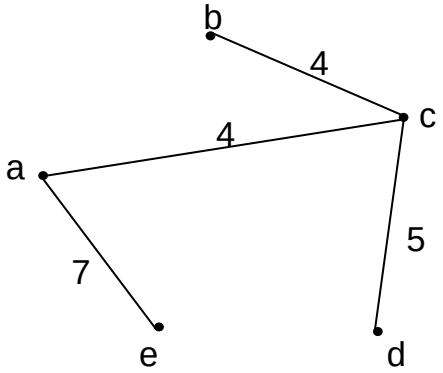
Ví dụ 5.12: Cho đồ thị G bao gồm 5 đỉnh a, b, c, d, e và độ dài các cạnh được cho trong hình 5.15.



Hình 5.15: Bài toán cây phủ tối thiểu

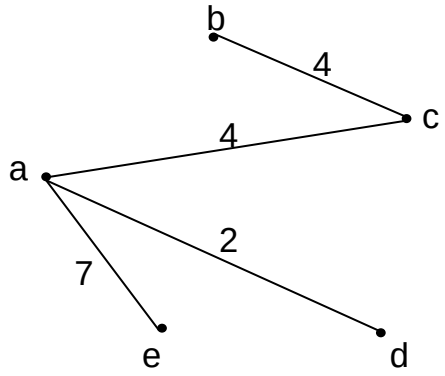
Tập hợp các cạnh để xét được thành lập theo thứ tự từ nhỏ đến lớn là ad, ab, be, bc, ac, cd, bd, de, ae và ce.

Cây xuất phát với giá là 20 (Hình 5.16).



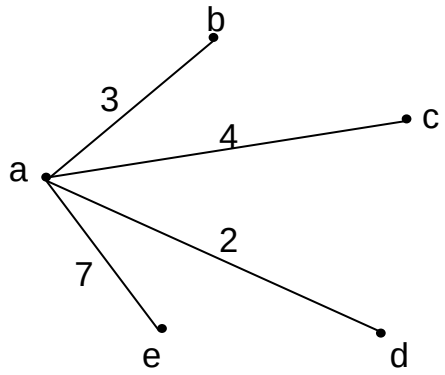
Hình 5.16: Cây phủ tối thiểu xuất phát

Thêm cạnh ad = 2, bỏ cạnh cd = 5 ta được cây mới có giá là 17 (Hình 5.17).



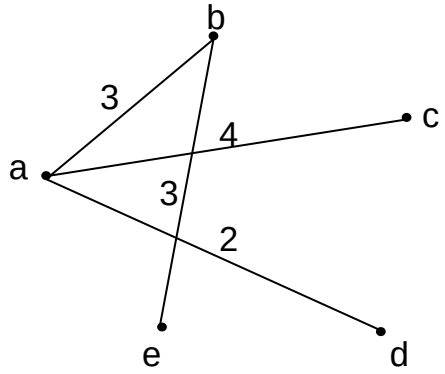
Hình 5.17: Cây giá 17

Lại thêm cạnh ab = 3, bỏ cạnh bc = 4 ta được cây có giá là 16 (Hình 5.18).



Hình 5.18: Cây giá 16

Thêm cạnh be = 3, bỏ cạnh ae = 7 ta được cây có giá là 12. (Hình 5.19).



Hình 5.19: Cây giá 12

Việc áp dụng các phép biến đổi đến đây dừng lại vì nếu tiếp tục nữa thì cũng không cải thiện được phương án.

Vậy cây phủ tối thiểu cần tìm là cây trong hình 5.19.

5.5.3 Bài toán đường đi của người giao hàng.

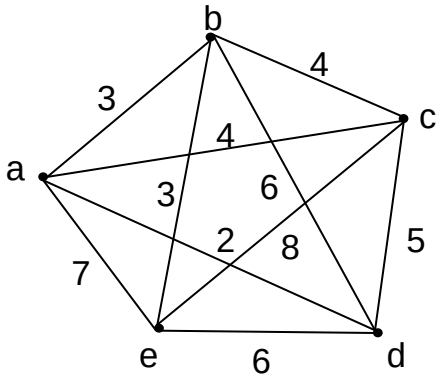
Ta có thể vận dụng kỹ thuật tìm kiếm địa phương để giải bài toán tìm đường đi ngắn nhất của người giao hàng (TSP).

• Xuất phát từ một chu trình nào đó.

• Bỏ đi hai cạnh có độ dài lớn nhất không kề nhau, nối các đỉnh lại với nhau sao cho vẫn tạo ra một chu trình đủ.

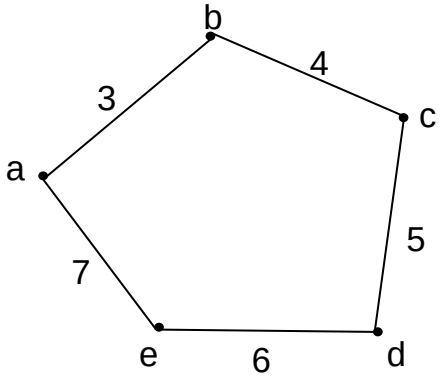
• Tiếp tục quá trình biến đổi trên cho đến khi nào không còn cải thiện được phương án nữa.

Ví dụ 5.13: Bài toán TSP có 5 đỉnh và các cạnh có độ dài được cho trong hình 5.20



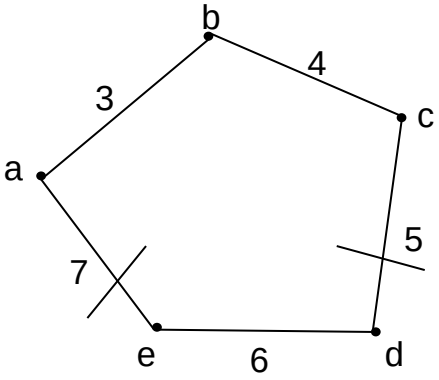
Hình 5.20: Bài toán người đi giao hàng

Phương án ban đầu là chu trình (a b c d e a) có giá (tổng độ dài) là 25. (Hình 5.21)

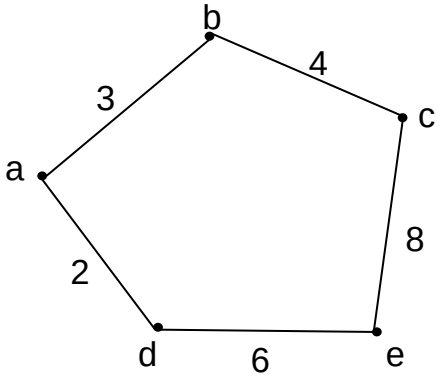


Hình 5.21: Phương án ban đầu, giá 25

Bỏ hai cạnh có độ dài lớn nhất không kề nhau là ae và cd (hình 5.22a), nối a với d và e với c, ta được chu trình mới (a b c e d a) với giá = 23 (Hình 5.22b).

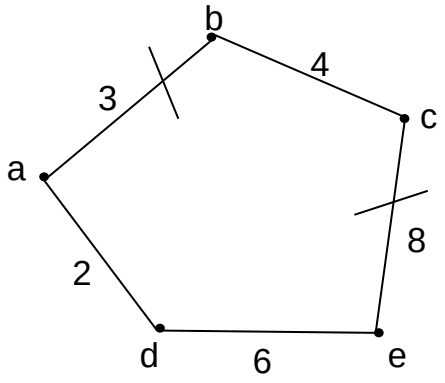


Hình 5.22a: Bỏ hai cạnh ae và cd

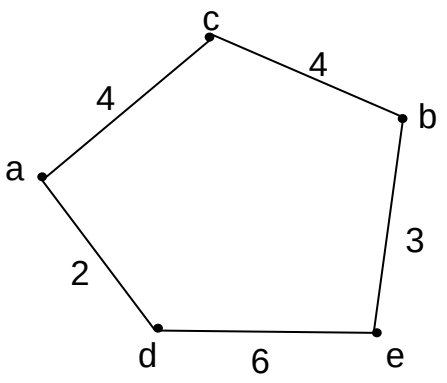


Hình 5.22b: Phương án mới, giá 23

Bỏ hai cạnh có độ dài lớn nhất, không kề nhau là ce và ab (hình 5.23a), nối a với c và b với e, ta được chu trình mới (a c b e d a) có giá = 19. (Hình 5.23b). Quá trình kết thúc vì nếu tiếp tục thì giá sẽ tăng lên.



Hình 5.23a: Bỏ hai cạnh ce và ab.



Hình 5.23b: Phương án mới, giá 19

5.6. TỔNG KẾT CHƯƠNG

Trong các kỹ thuật được trình bày trong chương, kỹ thuật chia để trị là kỹ thuật cơ bản nhất. Hãy chia nhỏ các bài toán để giải quyết nó!

Với các bài toán tìm phương án tối ưu, kỹ thuật "tham ăn" giúp chúng ta nhanh chóng xây dựng được một phương án, dẫu rằng chưa hẳn tối ưu nhưng chấp nhận được. Kỹ thuật nhánh cận cho phép chúng ta tìm được phương án tối ưu. Trong kỹ thuật nhánh cận, việc phân nhánh không khó nhưng việc xác định giá trị cận là điều quan trọng. Cần phải xác định giá trị cận sao cho càng sát với giá của phương án càng tốt vì như thế thì có thể cắt tỉa được nhiều nút trên cây và đo đó sẽ giảm được thời gian thực hiện chương trình.

Vận dụng phương pháp quy hoạch động có thể giải được rất nhiều bài toán. Điều quan trọng nhất để áp dụng phương pháp quy hoạch động là phải xây dựng được công thức đệ quy để xác định kết quả bài toán thông qua kết quả các bài toán con.

❖ **Bài tập củng cố:**

1. Giả sử có hai đội A và B tham gia một trận thi đấu thể thao, đội nào thắng trước n hiệp thì sẽ thắng cuộc. Chẳng hạn một trận thi đấu bóng chuyền 5 hiệp, đội nào thắng trước 3 hiệp thì sẽ tháng cuộc. Giả sử hai đội ngang tài ngang sức. Đội A cần thắng thêm i hiệp để thắng cuộc còn đội B thì cần thắng thêm j hiệp nữa. Gọi P(i,j) là xác suất để đội A cần i hiệp nữa để chiến thắng, B cần j hiệp. Dĩ nhiên i, j đều là các số nguyên không âm.

Để tính P(i,j) ta thấy rằng nếu i=0, tức là đội A đã thắng nên P(0,j) = 1. Tương tự nếu j=0, tức là đội B đã thắng nên P(i,0) = 0. Nếu i và j đều lớn hơn không thì ít nhất còn một hiệp nữa phải đấu và hai đội có khả năng 5 ăn, 5 thua trong hiệp này. Như vậy P(i,j) là trung bình cộng của P(i-1,j) và P(i,j-1). Trong đó P(i-1,j) là xác suất để đội A thắng cuộc nếu nó thắng hiệp đó và P(i,j-1) là xác suất để A thắng cuộc nếu nó thua hiệp đó. Tóm lại ta có công thức tính P(i,j) như sau:

P(i,j)	=	1	Nếu i = 0
P(i,j)	=	0	Nếu j = 0
P(i,j)	=	(P(i-1,j) + P(i,j-1))/2	Nếu i > 0 và j > 0

a. Viết một hàm đệ quy để tính P(i,j). Tính độ phức tạp của hàm đó.

b. Dùng kỹ thuật quy hoạch động để viết hàm tính P(i,j). Tính độ phức tạp của hàm đó.

c. Viết hàm P(i,j) bằng kỹ thuật quy hoach động nhưng chỉ dùng mảng một chiều (để tiết kiệm bộ nhớ).

❖ **Bài tập củng cố (tt):**

2. Bài toán phân công lao động: Có n công nhân có thể làm n công việc. Công nhân i làm công việc j trong một khoảng thời gian t_{ij}. Phải tìm một phương án phân công như thế nào để các công việc đều được hoàn thành, các công nhân đều có việc làm, mỗi công nhân chỉ làm một công việc và mỗi công việc chỉ do một công nhân thực hiện đồng thời tổng thời gian là nhỏ nhất.

a. Mô tả kỹ thuật "tham ăn" (greedy) cho bài toán phân công lao động.

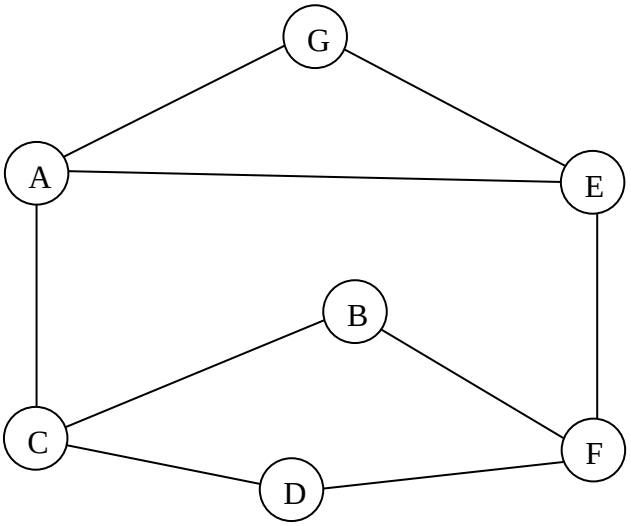
b. Tìm phương án theo giải thuật "háu ăn" cho bài toán phân công lao động được cho trong bảng sau. Trong đó mỗi dòng là một công nhân, mỗi cột là một công việc, ô (i,j) ghi thời gian t_{ij} mà công nhân i cần để hoàn thành công việc j. (Cần chỉ rõ công nhân nào làm công việc gì và tổng thời gian là bao nhiêu)

Công việc / Công nhân	**1**	**2**	**3**	**4**	**5**
1	5	6	4	7	2
2	5	2	4	5	1
3	4	5	4	6	3
4	5	5	3	4	2
5	3	3	5	2	5

3. Bài toán tô màu bản đồ thế giới

Người ta muốn tô màu bản đồ các nước trên thế giới, mỗi nước đều được tô màu và hai nước có biên giới chung nhau thì không được có màu giống nhau (các nước không chung biên giới có thể được tô màu giống nhau). Tìm một phương án tô màu sao cho số loại màu phải dùng ít nhất.

Người ta có thể mô hình hóa bản đồ thế giới bằng một đồ thị không có hướng, trong đó mỗi đỉnh biểu diễn cho một nước, biên giới của hai nước được biểu diễn bằng cạnh nối hai đỉnh. Bài toán tô màu bản đồ thế giới trở thành bài toán tô màu các đỉnh của đồ thi: Mỗi đỉnh của đồ thị phải được tô màu và hai đỉnh có chung một cạnh thì không được tô cùng một màu (các đỉnh không chung cạnh có thể được tô cùng một màu). Tìm một phương án tô màu sao cho số loại màu phải dùng là ít nhất



.

a Hãy mô tả kỹ thuật "tham ăn" (Greedy) để giải bài toán tô màu cho đồ thị.

b.Áp dụng kỹ thuật háu ăn để tô màu cho các đỉnh của đồ thị sau (các màu có thể sử dung để tô là: ĐỎ, CAM, VÀNG, XANH, ĐEN, NÂU, TÍM)

❖ **Bài tập củng cố (tt):**

4. Xét một trò chơi có 6 viên bi, hai người thay phiên nhau nhặt từ 1 đến 3 viên. Người phải nhặt chỉ một viên bi cuối cùng thì bị thua.

a. Vẽ toàn bộ cây trò chơi

b. Sử dụng kỹ thuật cắt tỉa alpha-beta định trị cho nút gốc

c. Ai sẽ thắng trong trò chơi này nếu hai người đều đi những nước tốt nhất. Hãy cho một nhận xét về trường hợp tổng quát khi ban đầu có n viên bi và mỗi lần có thể nhặt từ 1 đến m viên.

5. Xét một trò chơi có 7 cái đĩa. Người chơi 1 chia thành 2 chồng có số đĩa không bằng nhau. Người chơi 2 chọn một chồng trong số các chồng có thể chia và tiếp tục chia thành hai chồng không bằng nhau. Hai người luân phiên nhau chia đĩa như vậy cho đến khi không thể chia được nữa thì thua.

a. Vẽ toàn bộ cây trò chơi.

b. Sử dụng kỹ thuật cắt tỉa alpha-beta định trị cho nút gốc

c. Ai sẽ thắng trong trò chơi này nếu hai người đều đi những nước tốt nhất.

6. Cho bài toán cái ba lô với trọng lượng của ba lô W = 30 và 5 loại đồ vật được cho trong bảng bên. Tất cả các loại đồ vật đều **chỉ có một cái**.

Loại đồ vật	Trọng lượng	Giá trị
A	15	30
B	10	25
C	2	2
D	4	6
E	8	24

a. Giải bài toán bằng kỹ thuật "Tham ăn" (Greedy).

b. Giải bài toán bằng kỹ thuật nhánh cận.

TÀI LIỆU THAM KHẢO

❖ *TÀI LIỆU THAM KHẢO ĐỂ BIÊN SOẠN NỘI DUNG MÔN HỌC:*

- A.V. Aho, J.E. Hopcroft, J.D. Ullman; *Data Structures and Algorithms*; Addison-Wesley; 1983.

- Jeffrey H Kingston; *Algorithms and Data Structures*; Addison-Wesley; 1998.

- Đinh Mạnh Tường; *Cấu trúc dữ liệu & Thuật toán*; Nhà xuất bản khoa học và kỹ thuật; Hà nội-2001.

- Nguyễn Đức Nghĩa, Tô Văn Thành; *Toán rời rạc*; 1997.